



**MALMÖ
UNIVERSITY**

Faculty of Technology and Society

Department of Computer Science and Media Technology

Master Thesis Project 15p, Spring 2020

**Developing for Resilience:
Introducing a Chaos Engineering tool**

By

Ignacio Monge & Enikő Matók

Supervisors:

Magnus Krampell

Examiner:

Johan Holmgren

Contact information

Authors:

Enikő Matók

E-mail: matok.eni@gmail.com

Ignacio Monge

E-mail: ignacio753@gmail.com

Supervisors:

Magnus Krampell

E-mail: magnus.krampell@mah.se

Malmö University, Department of Computer Science and Media Technology.

Examiner:

Johan Holmgren

E-mail: johan.holmgren@mau.se

Malmö University, Department of Computer Science and Media Technology.

Abstract

Software complexity continues to accelerate, as new tools, frameworks, and technologies become available. This, in turn, increases its fragility and liability. Despite the amount of investment to test and harden their systems, companies still pay the price of failure. To withstand this fast-paced development environment and ensure software availability, large-scale systems must be built with resilience in mind. Chaos Engineering is a new practice that aims to address some of these challenges. In this thesis, the methodology, requirements, and iterations of the system design and architecture for a chaos engineering tool are presented. In a matter of only a couple of months and the working hours of two engineers, it was possible to build a tool that is able to shed light on the attributes that make the targeted system resilient as well as the weaknesses in its failure handling mechanisms. This tool greatly reduces the otherwise manual testing labor and allows software engineering teams to find potentially costly failures. These results prove the benefits that many companies could experience in their return of investment by adopting the practice of Chaos Engineering.

Keywords: chaos engineering, fault injection, resilience testing, distributed systems

Popular science summary

Breaking things on purpose: Introducing a tool to evaluate the resilience of software systems

From newspaper headlines to our own experiences accessing websites or using applications on our smartphones, digital users are aware of the fragile nature of software. A message that never arrived, a page that does not load or a ticketing system that crashed and the missed opportunity to buy those long-awaited concert tickets. Software is by nature inherently fragile and despite the efforts of engineering teams, large and small companies out there have shown to be susceptible to these unexpected and so-called bugs. In Sweden, a simple typo disturbed the access of all websites ending in .se for almost two hours. At the beginning of 2020, a software failure caused more than 100 flights to and from London's Heathrow airport to be disrupted.

In this thesis, a novel software tool is presented. Its purpose is to help build confidence in the resilience mechanisms of software systems to these types of bugs. It achieves so, by deliberately trying to break things. This means that a variety of failures and unexpected scenarios are injected into the system to prove its ability to cope with them – a concept known as chaos engineering. This thesis is aimed at researchers and practitioners who are interested in using this technique to test their systems. Moreover, these findings are relevant to the quality assurance processes of many software development companies.

These types of tools can be valuable instruments in the engineering team's toolbox to validate the capabilities of their systems to withstand failures. The results of this thesis shed light on some of the characteristics that make software systems resilient and how chaos engineering can be used to build confidence in them.

Acknowledgement

We are very grateful to our supervisor, Magnus Krampell, for his continuous support, valuable feedback, and his faith in us.

This thesis was made in collaboration with a company. We would like to thank all employees who helped and supported our project throughout the semester, especially our supervisor, Christina Schmidt.

Table of Content

1 Introduction.....	13
1.1 Historical Background of Chaos Engineering	15
1.2 Objective and Research Questions	16
2 Theoretical Background.....	17
2.1 Perturbation	17
2.2 Hypothesis	17
2.3 Experiment.....	17
2.4 Resilience Testing.....	17
2.5 Fault Injection.....	18
2.6 Chaos Engineering.....	18
2.7 The bigger picture of testing.....	19
3 Related Work	20
3.1 The Netflix Simian Army	20
3.1.1 Comment.....	20
3.2 Gremlin: Systematic Resilience Testing of Microservices.....	20
3.2.1 Comment.....	21
3.3 A Chaos Engineering System for Live Analysis and Falsification of Exception-handling in the JVM.....	21
3.3.1 Comment.....	21
3.4 Observability and Chaos Engineering on System Calls for Containerized Applications in Docker	22
3.4.1 Comment.....	22

4 Methodology	23
4.1 Activities of the Methodology	24
4.1.1 Activity 1: Problem identification and motivation	24
4.1.2 Activity 2: Define the objectives for a solution	24
4.1.3 Activity 3: Design and development.....	24
4.1.4 Activity 4: Demonstration.....	24
4.1.5 Activity 5: Evaluation	25
4.1.6 Activity 6: Communication.....	25
5 Results and Analysis	26
5.1 Activity 1: Problem identification and motivation	26
5.1.1 How to start chaos engineering.....	26
5.1.2 Principles of chaos	27
5.1.3 What to test	28
5.1.4 Motivation for practicing chaos engineering	28
5.2 Activity 2: Define the objectives for a solution.....	30
5.3 Activity 3: Design and development	32
5.3.1 The Target System	32
5.3.2 Functional requirements of the tool	32
5.3.3 Design & Architecture	34
5.4 Activity 4: Demonstration	44
5.4.1 Environment.....	45
5.4.2 Device	45
5.4.3 Metric.....	46
5.4.4 Description of the experiments	46
5.5 Activity 5: Evaluation.....	51
5.5.1 Objective 1: Minimal instrumentation of the target system.....	51
5.5.2 Objective 2: The tool introduces chaos at the message passing level.....	51
5.5.3 Objective 3: Attacks can be targeted with a variable probability	52

5.5.4	Objective 4: The tool can be stopped at any time.....	53
5.5.5	Objective 5: The tool grants observability of the scope of the chaos	53
5.6	Activity 6: Communication	54
6 Discussion and Conclusion.....		55
6.1	Answering Research Questions	55
6.1.1	RQ1: What is a reasonable methodological approach to develop a chaos engineering tool?	55
6.1.2	RQ2: What are the functional requirements of a chaos engineering tool for the target system?	55
6.1.3	RQ3: How can a chaos engineering tool be designed and architected for the target system?	56
6.1.4	RQ4: What are some of the attributes that make a system resilient and how can chaos engineering be used to build confidence on them?.....	57
6.2	Contribution to Chaos Engineering	58
6.3	Contribution to Software Testing	58
6.4	Threats to validity	59
7 Future Work.....		60
7.1	Supporting other message passing channels and content types.....	60
7.2	Improve user control over attacks.....	60
7.3	Automation	60
Appendix I: Endpoint mapping		61
Appendix II: Chaos Tool Properties.....		62
Appendix III: Experiments Description.....		64
1.	Introducing Latency Experiment	64

2.	Modify Field Experiment	66
3.	Remove Field Experiment	70
4.	Add Field Experiment	78
5.	Fail Request Experiment.....	80
6.	Creating High Memory Usage Experiment	83
7.	Creating high CPU usage Experiment.....	86
8.	Stop all running attacks Experiment.....	89
References		92

List of Figures

FIGURE 1. METHODOLOGY [15]	23
FIGURE 2. COMMUNICATION CHANNELS OF THE TARGET SYSTEM	32
FIGURE 3. PROXY APPROACH	35
FIGURE 4. DEPENDENCY APPROACH.....	35
FIGURE 5. MEMORY CALCULATIONS IN THE JVM	84

List of Tables

TABLE 1: OBJECTIVE-REQUIREMENT MAPPING	32
TABLE 2: EXPERIMENTS DESCRIPTION	47

List of acronyms

AOP	Aspect Oriented Programming
API	Application Programming Interface
AWS	Amazon Web Services
DNS	Domain Name Server
DSRM	Design Science Research Methodology
GDPR	General Data Protection Regulation
GUID	Globally Unique Identifier
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
MNO	Mobile Network Operator
OOP	Object-Oriented Programming
PaaS	Platform-as-a-Service
PII	Personally Identifiable Information
REST	REpresentational State Transfer
XML	Extensible Markup Language

1 Introduction

Modern web applications are becoming increasingly complex. The rate at which software development is done, as well as the growing variety of programming languages, has changed the way we develop software. Systems must be built to support rapid changes and ensure timely responsiveness to user feedback [1]. On the other hand, intrinsic to the software development nature, is the fragility of systems [2]. In this context, large-scale systems must be built to foresee and mitigate many software failures [3].

Using distributed cloud systems has become a popular solution for building and developing applications in software companies nowadays [1]. Even though the approach is promising, development teams still face challenges when managing and operating these systems. Some of these problems can be failures caused by dependencies [1], a sudden surge in requests coming from clients [3], or unreliable network causing latency issues [1].

Despite the best efforts of engineering teams, who strive to prevent, detect, and repair software bugs, large scale systems still fail and sometimes even catastrophically [1]. For example, a failure in Amazon Web Services in 2011 caused several websites platforms such as Reddit, Quora, and Foursquare among many others to crash [4]. In Sweden, a Domain Name Server (DNS) bug took almost a million websites off the Internet [5].

Companies invest large amounts of resources into testing, hardening, and overall good engineering practices to develop highly available systems [2]. However, as shown in the previous examples, despite those efforts, most software applications remain prone to bugs [2]. This can be due to poor training, technical debt, or management pressure to deliver code swiftly at the expense of quality [2]. For this reason, it is becoming increasingly essential for developers to test and measure the resilience of their systems [6].

Chaos Engineering is a software development practice that aims to address some of these challenges by deliberately injecting so-called perturbations into the system [6]. By analyzing and identifying the collateral effects, developers can build confidence over the resilience of their systems. This is achieved by specifying and evaluating hypotheses. First, assumptions about the fault tolerance of the system are formulated. Then, faults are injected into the production system, and their impact is observed. Finally, new knowledge on the strengths and weaknesses of the system is created [6].

Adopting chaos engineering also comes with some cost: the engineering effort it requires to implement it and to mitigate the intentionally and accidentally caused harm [7]. That is why when first faced with the idea, it is quite common for IT departments to only see the risk of chaos engineering without analyzing the potential benefits [8]. Therefore, it is important to clearly analyze if the business goals align with the chaos engineering practices and whether the cost of the impact is worth taking the chance [7].

On the other hand, many benefits can be listed when practicing chaos engineering. From the customers' point of view, the availability and reliability of the service will improve, and the number of outages will decrease [7]. By having chaos tests, the engineers will be able to reduce the number of preventable disruptions in the system [7] and the development team can put a bigger emphasis on building fallback and recovery logics. Overall, companies can see many positive effects in their returns of investments by adopting this practice.

1.1 Historical Background of Chaos Engineering

The concept of chaos engineering can be described as unconventional, but it's far from new. Back in 1975, there was a proposal to introduce fake “ghost planes” in an air traffic control system [2]. The idea, initially suggested by Yau and Cheung, reasoned that if the air traffic controllers could safely land all airplanes (including the ghost ones) the system would be deemed trustworthy [2]. This concept would be then referred to as failure injection.

Until very recently, this idea of deliberately introducing failure remained relatively unheard of. It was in 2011, when Netflix released a post in their official tech blog [9], that the term was brought back to light in the context of software development. The article was titled “The Netflix Simian Army” [9], and it is here where the term “Chaos Engineering” was first coined.

Chaos engineering is a scientific method to verify resilience hypotheses about software systems [10]. In the case of Netflix, they hypothesize that their systems are resilient to server crashes. Using a ‘chaos experiment’ the team can verify this. For instance, the “Chaos Monkey” randomly crashes production servers, and the “Latency Monkey” arbitrarily increases and decreases the latency in the server network [10] [9]. At the same time, the team measures that the number of video streams per second (a key business metric) remains unchanged [6].

Eventually, companies like Microsoft [11], Facebook [12], and Amazon [8] joined the movement, by publicly discussing their chaos engineering practices. For instance, Microsoft's Azure search team has its own Search Chaos monkey, which they also discussed in a tech blog post [9]. In the case of Facebook, they have their own chaos system called Project Storm, which they had been working on since 2012 [12] but was only openly discussed in an interview with Forbes in 2016 [12].

The term has evolved ever since, thanks to industry efforts to capitalize on it, with commercial tools becoming increasingly available. Despite the fact that the academic literature on this topic is very scarce, there is an increasing interest from the community and industry sector, which reveals the shift in focus that these players are putting on this particular engineering practice [2].

1.2 Objective and Research Questions

In this thesis, the design, implementation, and evaluation of a chaos engineering tool are presented. Existing chaos engineering efforts are analyzed with the purpose of better understanding failure in distributed software systems and the role that chaos engineering can play to mitigate it. More specifically, the objective of this work is to create a better understanding of the key factors that make chaos engineering a suitable option for testing a system's reliance.

The research questions for the thesis are:

RQ1: What is a reasonable methodological approach to develop a chaos engineering tool?

By answering this question, a better understanding is created on how to develop a chaos engineering tool.

RQ2: What are the functional requirements of a chaos engineering tool for the target system?

In order to identify relevant requirements and features for a chaos engineering tool, the target system needs to be assessed – hence it is important to understand its architecture and technology stack.

Additionally, best practices and practitioners' learnings have to be considered to determine a standard set of functionalities that chaos engineering tools should have.

RQ3: How can a chaos engineering tool be designed and architected for the target system?

In this research question, the technical solutions and system architecture that fit the system to be tested are described.

RQ4: What are some of the attributes that make a system resilient and how can chaos engineering be used to build confidence on them?

This question aims to collect some attributes that characterize a resilient system. With the use of a chaos engineering tool, these attributes can be tested and challenged. As an outcome of a chaos test, it can be concluded that either the feature is resilient or that it needs to be improved. Hence the effectiveness of the tool is verified.

2 Theoretical Background

This section gives a brief overview of the important key concepts mentioned in the report in order to proceed with a better understanding of the topic.

2.1 Perturbation

In the field of chaos engineering, a perturbation occurs when the execution flow, state, or environment of the software is changed on purpose in a controlled way [6]. The impact of the perturbation is observed, and new conclusions can be drawn on the resilience of the system [6].

2.2 Hypothesis

Hypotheses are the relationships between monitored/observed behaviors and controlled perturbations [6]. The number of combinations of failure scenarios that can be tested on a distributed system's architecture is exponential, therefore creating and formulating valid hypotheses that strive for detecting maximum failure is essential to guarantee the efficiency of the chaos engineering tool [3].

2.3 Experiment

A chaos experiment aims to either validate or falsify a hypothesis made about a perturbation [6]. It includes the injection of the perturbation and the process of monitoring how the system under test reacts. Common approaches are to either randomly or deterministically inject failure in the systems [1] [3] [6].

2.4 Resilience Testing

Resilience testing begins with accepting the fact that systems will fail no matter what [8]. This kind of system testing aims to make sure that the developers work in a manner of eliminating single points of failure. The goal is to trigger these failures on the desired terms and expose the software's defects [8]. These triggered failures can either be individual or system failures.

Resiliency testing needs to be feedback-driven and systematic [1]. This means that the operations should orchestrate the triggering of the failure and obtain quick feedback about

the test's results. It is also worth mentioning that the authors of [8] refer to Erik Hollnagel's principles of resilience:

*“(1) know what to expect (anticipation);
(2) know what to look for (monitoring);
(3) know what to do (how to respond);
and (4) know what just happened (learning)” [8].*

2.5 Fault Injection

The traditional way to practice fault injection is by setting up an offline test environment and creating targeted perturbations [6]. These tests aim to find bugs in a specific feature and gain information about the failure-handling capabilities.

2.6 Chaos Engineering

As Netflix's engineers define it:

“Chaos Engineering is the discipline of experimenting on a system in order to build confidence in the system's capability to withstand turbulent conditions in production.”
[10]

Chaos engineering is a new and emerging field of resiliency testing in production, that consists of trying to break things on purpose and observing the system's reaction and recovery mechanisms. Chaos engineering focuses on investigating hypotheses created about the system's resiliency [6].

By observing the system's fault tolerance developers can either gain confidence in their code or reveal hidden points for failure in the system. These tests are run in a controlled manner and the experiments aim to gather new information about the system's resiliency and availability features. Running these tests in a controlled way is a much lower cost to pay for learning about the software's vulnerabilities than having the problems surface on their own [8]. During a chaos experiment, the developers can observe if the system can fail gracefully and can offer partial availability with the implemented fallbacks [13]. Even with implemented fallbacks, a non-critical service's failure can lead to a system-level disruption due to cascading failures [13].

2.7 The bigger picture of testing

Unit tests are used to address the validity of the individual components [7]. The unit gets an input and a specific output is expected. The next step is conducting integration tests which attempt to find out whether the components work together seamlessly [7]. It is followed by system testing, which consists of evaluating the complete system based on the specified requirements. Finally, chaos experiments come into the picture when engineering teams try to validate assumptions on the components' ability to withstand failures.

3 Related Work

In this section papers, journal articles, and other resources that are related to the field of Chaos Engineering are presented and summarized. Specifically, those describing efforts to create such systems were selected and analyzed. Furthermore, the comment subsection offers insights as well as highlights any relevant information.

3.1 The Netflix Simian Army

In a technical blog post [9], Izrailevsky and Tseitlin presented for the first time Chaos Engineering and the Simian Army [9]. A collection of standalone tools, each of them implementing a singular chaos feature. The first and most well-known, Chaos Monkey randomly shuts down AWS instances in the Netflix cloud [9]. After the effectiveness of Chaos Monkey was confirmed, the creators went on to develop more chaos tools such as Janitor Monkey, a program that looks for unused resources in the system and disposes them [9], or Chaos Gorilla, a tool that shuts down an Amazon availability zone [9].

3.1.1 Comment

Shutting down instances is one of the most classical examples of chaos engineering, as this was the first approach taken in this area. This is a path explored by many practitioners

3.2 Gremlin: Systematic Resilience Testing of Microservices

Heorhiadi et al. [1] developed a chaos tool that works over microservices architectures. Gremlin builds on top of the message-exchanges patterns that these services rely on when communicating over the network [1]. The tool supports the emulation of crash, performance, and fail-stop failures. These types of failures are observable from a network perspective by other microservices [1].

3.2.1 Comment

This paper lacks several aspects of research design, the methodology used, among others that hinder its validity. Gremlin is a commercial tool, which could be an explanation for the lack of verifiable data. Nevertheless, its status as a paid product proves its viability as a chaos engineering solution.

This paper points to a particular direction: how to develop chaos systems that do not require prior knowledge of the system internals as well as modifications of the source code. This is a particular research area that could be further explored.

3.3 A Chaos Engineering System for Live Analysis and Falsification of Exception-handling in the JVM

One example of a chaos engineering tool that has been developed through research is Chaos Machine. In their paper, Zhang et al. [6] argue that the core novelty of their system is that it considers “error-handling capabilities at the fine-grained level of programming language exception” [6]. The tool is capable of exposing the system’s strengths and weaknesses for every executed try-catch code block. These blocks are the places where the exception handling logic is performed in high-level languages such as Java. The system works by injecting exceptions in try-catch blocks and measuring the system’s performance.

The system was only tested against three large scale open-source Java applications, despite the fact that the system is able to run any language that compiles to Java Virtual Machine (JVM) [5], like Kotlin or Scala. The authors argue that their machine is the only one that does chaos engineering at the level of exception handling and try-catch blocks [6].

3.3.1 Comment

One particular aspect of this research is the fact that the authors also measured the overhead of the Chaos Machine in the computer resources [6]. Good chaos engineering systems should cause as little overhead on the host systems as possible. Therefore, minimizing its running costs is one of the key characteristics that practitioners look for in these types of tools.

3.4 Observability and Chaos Engineering on System Calls for Containerized Applications in Docker

Simonsson et al [14] proposed Chaos Orca, a novel fault injection tool for system calls in containerized applications [14]. One of the key innovations of their system is its ability to conduct experiments without instrumenting the application [14]. To achieve this, it introduces bugs or delays in the execution of systems calls [14].

One differentiating feature of this system implementation is how the host application is treated as a black box. This allowed the researchers to develop a tool that is agnostic of the internal implementation of the host application. This turn raised the question of how to measure changes in the system if the application cannot be directly observed. One way the authors came about this, was by monitoring the surroundings for impact on physical resources [14].

3.4.1 Comment

As previously mentioned, this paper along with [1] approaches the system under test as a black box, providing a system-agnostic tool. This means that the tool is able to conduct experiments without the need to understand the internal mechanisms or modifying the host application. Furthermore, the authors engineered a way to monitor the surroundings of the system to measure the effects of testing on the resources. This is a key insight to consider when developing a chaos tool.

4 Methodology

For this thesis, the methodology proposed by Peffers et al [15], called Design Science for Research Methodology in Information System (DSRM) is used [15]. Design Science focuses on the creation and evaluation of IT artifacts that can solve organizations' problems [15]. This is achieved through a rigorous process that involves the design, evaluation, and communication of the results of the artifact [15].

This methodology follows an iterative approach for system development. This means that it allows using previous steps and iterations as feedback loops that help to enhance the system by building over the final result (Figure 1). Moreover, this methodology provides four entry points (shown as circles in Figure 1). For this thesis the problem-centered initiation was used to begin the process. This allows to start the thesis by identifying the problem and motivating the research.

Furthermore, DSRM provides a framework where all the thesis' research questions can be answered. Its methodological approach provides a reasonable method as an answer to the first research question. Meanwhile, the third step tackles the second and third research questions regarding the development of the tool. Finally, the demonstration and evaluation steps aim at the fourth question, by providing some insight into the resilience of the target system.

With a chaos tool as one of the pivotal artifacts of this research, this methodology allows to naturally develop and assess its effectiveness at generating chaos. The methodology's final artifact will help build confidence in the target system's resilience, addressing the main goal of the research.

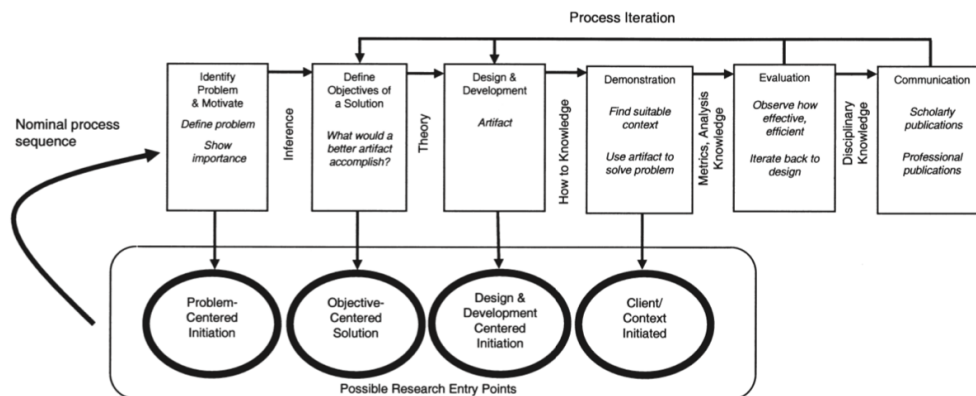


Figure 1. Methodology [15]

4.1 Activities of the Methodology

In this section, the steps (shown as boxes on Figure 1), or so-called activities of the methodology are presented and described. DSRM is composed of six activities which are: problem identification and motivation, definition of objectives for a solution, design & development, demonstration, evaluation, and communication.

4.1.1 Activity 1: Problem identification and motivation

In this step, the research problem is defined, and the value of a solution is justified. This activity helps to motivate the research and build a common conceptual understanding [15]. It includes gathering knowledge on the state of the problem as well as the importance of a solution [15].

4.1.2 Activity 2: Define the objectives for a solution

The solution objectives are inferred from the problem definition in the previous activity. They describe how the artifact addresses problems not previously tackled and how the proposed solution is better than existing ones [15].

4.1.3 Activity 3: Design and development

This step involves the actual creation of the artifact. In this activity, the desired functionality and its architecture are defined [15]. The system architecture works as a blueprint that guides the development of the tool. This step also helps identify the relationships and interactions between the components of the system.

4.1.4 Activity 4: Demonstration

In this activity, the artifact is used to demonstrate how it can solve different instances of the problem. Experiments are conducted in the target system to determine the usefulness of the tool to address the problem [15]. As a result of this activity, data is gathered in order to assess the resilience of the target system. Furthermore, as a by-product, all the functional capabilities of the tool are tested in the experiments.

4.1.5 Activity 5: Evaluation

This step involves observing and measuring the artifact's performance against the previously defined problem. By comparing the objectives of the solution defined in activity 2 to the actual observed results it can be decided to further iterate to activity 3 or to leave any potential improvements to subsequent projects [15]. The target system is measured through quantifiable business and operational metrics. For this step, empirical evidence from the experiments is collected with relevant metrics.

4.1.6 Activity 6: Communication

The problem and its importance, as well as the artifact and its novelty, are communicated to the community [15]. The result of this project is a tangible IT artifact in the form of a chaos engineering tool. By analyzing the findings and observations some conclusions can be drawn on the features that make systems resilient. This thesis will be publicly available in Malmö University's Electronic Publishing platform (MUEP). The presented chaos engineering tool, as well as the findings, will serve future researches, and practitioners continue to develop knowledge in a field where there is still room for research to be done.

5 Results and Analysis

In this section, the results for this thesis are presented following the activities described in section 4 [15].

5.1 Activity 1: Problem identification and motivation

The first activity of the methodology consists of identifying the problem and creating a better understanding and motivation to solve it. As a part of this, a literature review was conducted.

The literature review explores existing best practices on how to start chaos testing as well as identifying the value of doing chaos testing by looking at some of the benefits of adopting this practice.

For this purpose, the databases of IEEE Explorer, ACM digital library, and Malmö University digital book collection were all queried in the search process.

Given the novelty nature of this topic and the role that industry has played in this emerging field, as the first players to put forward this term, several online articles and resources were also considered. For this same reason, the search terms used in this exploration were expanded beyond “chaos engineering” to also include similar definitions found throughout the literature such as “fault injection” and “resilience testing”.

5.1.1 How to start chaos engineering

When a company starts to adopt chaos engineering, they need to begin small [16]. Conducting the first chaos experiment in production is risky, therefore the authors of [17] propose setting up a dedicated test environment and having small scale failures injected in the beginning. As stated in the principles of chaos by Netflix [10], if confidence in the chaos system is established, it can then be run in production in an automated manner.

For creating an experiment, the engineers of Netflix established these four steps [10]:

1. Identify a metric that indicates the system’s normal behavior

In order to assess the system’s normal behavior, it is possible to define a variable that combines both operational and user experience metrics [10]. For example, a successfully rendered page or the number of transactions per second.

2. Create a hypothesis about the metric not changing during perturbation

At this step, there is a need to discuss the bottlenecks of the system and think about the “what if” scenarios in order to create the correct hypothesis [6].

3. Design the chaos experiment and run it

When designing a chaos experiment it is crucial to properly scope them and identify those metrics that are worth measuring during the test [10]. It is also important to accurately notify the organization before running the experiment.

4. Evaluate the hypothesis

In this step, the chaos test either validates or falsifies the hypothesis formulated in step two. After the experiment, the team should document the outcomes and avoid blaming individuals for the discovered fallacies [10].

5.1.2 Principles of chaos

The principles of chaos are an empirical, system-based approach addressing chaos testing in distributed systems at scale [18]. They are developed by the chaos community, a group of practitioners building resilient systems through chaos engineering [18]. Furthermore, they allow to uncover systemic weaknesses and to build confidence in the ability of the systems to withstand real-world conditions [18]. These principles come to encompass and solidify some of the concepts presented above. Some of the principles are:

a) Build a Hypothesis around Steady-State Behavior

As previously discussed, when creating hypotheses, the focus should be on a measurable output of the system under normal behavior. The system’s throughput, error rates, could be metrics of interest [18]. This steady-state should continue in both the control and experimental set of tests.

b) Vary Real-world events

Chaos testing should reflect real-world scenarios, therefore priority should be given to events based on their impact or frequency [18]. For example, malformed requests/responses or events like sudden increments in traffic. Any event with the capacity to disrupt a steady state is a good candidate for a chaos experiment [18].

c) Minimize Blast Radius

The blast radius is the distance from the source that will be affected when an explosion occurs [18]. In chaos engineering, this means the reach of the failure caused by experiments. While there must be a space for some short-term impact, the customers, clients, and even the company itself should not be exposed to unnecessary negative effects. This principle ensures that the fallout from experiments is contained [18].

5.1.3 What to test

In this subsection, a few ideas collected in the literature review about possible test scenarios are presented.

a) Injecting artificial latency into the communication [10]

As the authors of [1] mention the communication between microservices in the cloud happens over the network. Therefore, these standard application protocols (e. g. HTTP) can be used to introduce latency.

b) Fail requests between services [10]

Failing some requests in the communication between the services can reveal problems with retries, timeouts, and circuit breaker strategies [7].

c) Maxing out CPU or memory capacity [17]

Developers mostly focus on software bugs and do not take hardware-caused failures into consideration. Simulating them can be used as a type of failure injection.

5.1.4 Motivation for practicing chaos engineering

It is not unusual for IT departments to only see the risks and costs associated when first faced with the idea of adopting chaos engineering [8]. However, as claimed by practitioners, once chaos engineering has been adopted, the mindset quickly shifts. The team focuses on the opportunity to learn something new and fix the broken code [8]. The software engineers start to design for failure and create systems that are more fault-tolerant and have recovery logics [11].

The engineers of Netflix [10] also claim that practicing chaos testing makes their systems stronger and gives them confidence in their reliability. With controlled

experiments, they can easily uncover bugs that would have otherwise led to significant damages [7].

Furthermore, the benefits of chaos engineering can be categorized at the customer, business, and technical level [19].

a) Customer

From the customers' point of view, the availability and reliability of the service will improve, and the number of outages will decrease. By having chaos tests, the engineers will be able to reduce the number of preventable disruptions in the system [7].

b) Business

Practicing chaos engineering can come with business risks (data corruption, financial losses, and partially degraded user experience [6]). However, these can be much higher if a sudden outage of the services happens in production. For instance, the developer team might not be used to the tasks of stabilizing the system, and the software could lack fallback mechanisms [16]. By adopting chaos engineering, the development team can put a bigger emphasis on building these logics.

Having automated chaos tests in the system will make the developers' everyday routine to fix things when they break. These chaos tests can become part of the work culture and make the teams more cohesive and engaged, by adopting a non-blame culture [8]. As an engineer of IBM puts it, having a more resilient system strengthened by chaos tests “will also keep you out of tomorrow's headlines” [16].

c) Technical

From a technical perspective, the insights gained from chaos experiments can reduce the number of incidents and decrease the time to recover from outages. These experiments will also help the developers to better understand the whole system and its failure mechanisms. The tests can reveal classical architecture defects and bad design choices. For example, some latent defects only appear when a failure is triggered [19]. The experiments will also highlight and create a better understanding of the criticality of each service [7]. Overall, the general stability of the production system will improve and the time for recovering from disruptions can be greatly reduced [16].

5.2 Activity 2: Define the objectives for a solution

The solution objectives are defined following the learnings gathered in activity 1 (section 5.1). This includes the principles of chaos engineering [18], as well as the research gaps found in the existing literature. These insights allow to define the main goals of the tool, which will serve as the blueprint for the functional requirements in the next activity.

Objective 1: Minimal instrumentation of the target system

Previous practitioners' and researchers' tools point to the benefits of having a tool unaware of the internal logic of the target system [1][14].

This objective decouples the tool from the system to be tested, allowing to generalize the results and to extend the usefulness of the tool to other systems with similar characteristics.

Objective 2: The tool introduces chaos at the message passing level

The principles of chaos lay out the importance of prioritizing real-world events [18]. This objective aims to disrupt the state of the application at the message passing level where the interactions in a distributed system occur. Furthermore, the tool aims to produce chaos following the ideas presented in section 5.1.3. For instance, adding latency, failing requests, or maximizing the use of resources. This helps to assess the tolerance capabilities of the target system at the message passing level.

Objective 3: Attacks can be targeted with a variable probability

One of the most common approaches when doing chaos engineering is to randomly inject failure in the systems [1] [3] [6] [14]. While one of the advantages of this approach is its simplicity and generality [3] its main drawback is that it cannot shed light on deeper and more intricate issues.

This objective gives a more knowledgeable developer the ability to target a perturbation to a particular point of the system, with the certainty that it will be injected. At the same time, it allows the developer to experiment with randomly injected perturbations when needed.

Objective 4: The tool can be stopped at any time

The normal operability of the target system is a priority and chaos experiments should be easily stopped at any given time. This objective guarantees compliance with the principles of chaos engineering by providing a functionality that can help minimize the blast radius of the experiments [18]. As a result of this, any negative impact can be contained and limited to a given period of time.

Objective 5: The tool grants observability of the scope of the chaos

This objective complies with the principles of chaos, by providing an observation mechanism that helps to monitor the reach of the chaos experiments between control and experimental groups [18]. When the target system is under attack the results of the experimental groups contain extra information regarding the type of attacks that are being subjected to, their state, and the targeted system location. This information helps to debug and report any failures that might occur.

5.3 Activity 3: Design and development

In order to answer the second and third research questions, in this activity, the target system is introduced. Then the identified functional requirements are presented along with the design and architecture of the tool.

5.3.1 The Target System

The target system needs to be assessed in order to define a suitable design and architecture for the chaos engineering tool. This system is a Java-based Spring Boot application developed and maintained by a large telecommunications Platform-as-a-Service (PaaS) company. It provides an API (Application Programming Interface) in the cloud for the company’s clients, who want to send text messages to customers’ cell phones. The API can be engaged via RESTful services (REpresentational State Transfer), meaning that the exposed services can be accessed over the web. The target system also interacts with Mobile Network Operators (MNO) who allow the distribution of the clients’ intended messages. Figure 2. illustrates the two channels of communication of the target system



Figure 2. Communication channels of the target system

5.3.2 Functional requirements of the tool

This section focuses on the necessary requirements. A requirement is a feature or capability of the tool. These are defined based on the general objectives listed in the previous activity, and they can be mapped as follows (Table 1):

Table 1: Objective-Requirement mapping

OBJECTIVE NO.	REQUIREMENT
1	Requirement 1
2	Requirement 2 and 3
3	Requirement 4
4	Requirement 5
5	Requirement 6

Requirement 1: Minimal instrumentation of the target system

This requirement guarantees that no logic is leaked over to and from the target system in order to make the chaos tool functional. It ensures a clear separation between the chaos tool and the target system. This means that no changes should be made into any of the existing logic or source files, beyond those required to install the tool.

Requirement 2: The tool produces chaos at the message passing level

The tool is able to intercept messages to inject attacks. Interactions at the message passing level include HTTP and RESTful requests as well as public method calls between classes.

The list of attacks is:

- Introducing latency
- Changing the contents of the HTTP messages
 - Adding, removing and modifying the contents of HTTP messages
- Failing requests
- Creating high memory or CPU usage

Requirement 3: The tool can target different points of message passing interaction

This requirement introduces support for disturbing different message-passing interactions in the target system. An attack can be targeted to a RESTful controller where incoming requests are being processed, or to a class' public methods where the interaction between components is held.

Requirement 4: Attacks can be targeted using a probability property

Once the tool is active, the user will be able to set a probability property that will decide via random distribution, if the perturbation should be injected or not. This requirement guarantees that the third objective is met by controlling the occurrence probability of the attacks.

Requirement 5: The tool provides a functionality to stop running attacks

In order to prevent any further breakdown of the target system and recover its normal operational state, the chaos tool provides a functionality that allows to completely halt running perturbations.

Attacks running in self iterating loops are short-circuited with a signal and stopped. Moreover, no new perturbations can be introduced as a result of activating this functionality.

Requirement 6: The tool provides standard output logging around chaos events

Logging will be used to show when the system is under experimentation. Hence one of the requirements of the tool is to be able to provide logs around chaos events. The standard output is used for this purpose, and logging messages are added before and after a perturbation occurs.

The log messages describe the actual perturbation that is being initiated or finalized. Additionally, the tool logs the method or function being targeted as well as the input parameters.

5.3.3 Design & Architecture

With an understanding of the target system and the required features, the design, and architecture of the chaos tool that meets each of the functional requirements are presented.

5.3.3.1 Requirement 1: Minimal instrumentation of the target system

In order to address this requirement, the tool was built using the so-called man-in-the-middle approach and leveraging the technical solution provided by Spring Boot Starter Applications. In the following subsections, these concepts are further elaborated.

5.3.3.1.1 *Man-in-the-Middle*

The man-in-the-middle approach describes a technique that allows to insert an actor as a proxy into the communication session between two components or systems [20]. By using this technique two of the main requirements for the tool can be satisfied: minimal instrumentation of the target system (requirement 1) and to produce chaos at the message passing level (requirement 2).

One way to implement this technique would be to create a separate system that acts as a proxy between the clients and the target system (shown in Figure 3). A disadvantage of this approach is that additional infrastructure and resources are required to set up a server that can host the tool. It would also require additional modifications so that the traffic is routed between the proxy and the interacting parties. Furthermore, this approach only allows intercepting messages at the HTTP level, leaving out the possibility to intercept other types of messages, as stated by the second requirement.

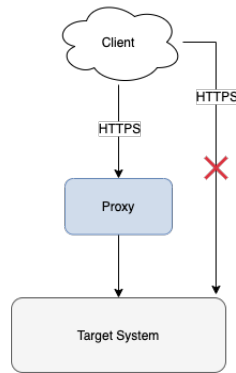


Figure 3. Proxy Approach

Hence, the solution implemented by the tool leverages the knowledge of the target system, by creating a plugin library that can be added (as shown in Figure 4) and used to intercept the requests. By using the same technology as the target system, different programming techniques that allow intercepting incoming requests, as well as other public method messages, can be used without the need to create additional server resources. These techniques are explained in further detail in the description of the implementation of the second (5.3.3.2) and third requirement (5.3.3.3) subsection.

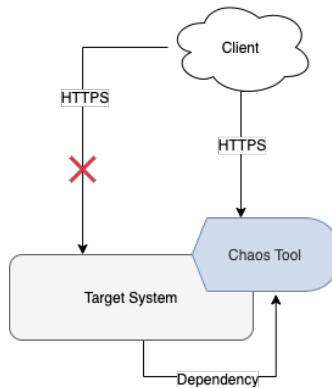


Figure 4. Dependency Approach

Another advantage of this approach is that it still keeps the chaos tool decoupled from the internal implementation and logic details of the host system. By creating a plugin component, the tool's code can be kept separate from the target system. The tool would only need to be added as a dependency of the target system to work. In order to achieve this the Spring Boot Starter dependency management solution is used. An explanation of this technology is provided in the following subsection.

5.3.3.1.2 *Spring Boot Starter Application*

A Spring Boot Starter application is a module that can be added to other Java-based Spring Boot projects as a Maven or a Gradle dependency [21].

Via automatic configuration capabilities, it can run as an addition to the “host” application [21]. Autoconfiguration works by having a Java class annotated with `@Configuration`, then listing it in a resource file called “spring.factories” [22]. This file is then read by the host application, which looks for the configuration files to populate the Spring application context [22].

A module developed to be a Spring Boot Starter application needs a specific build configuration, meaning that through its build process an executable jar should not be produced, but rather just a jar library [21].

This chaos tool leverages the concept of Spring Boot Starters [21] to fulfill the first requirement.

5.3.3.2 Requirement 2: The tool produces chaos at the message passing level

The tool implements a variety of different attacks at the message passing level. In this section, each attack is described in detail along with its technical implementation. The attacks have been grouped into three categories based on their similarities and the programming techniques used to implement them.

5.3.3.2.1 *Introducing Latency & Fail Request*

The attacks that introduce latency and failing requests simulate scenarios where the HTTP requests are delayed due to inconsistencies of the network or unresponsive servers.

Both attacks work by using a concept inside Spring Boot known as interceptors [23]. An interceptor catches all the incoming requests for REST controller classes. This technique allows executing any type of logic before the endpoint gets the request [23].

The latency attack works by artificially introducing latency into the incoming requests. Latency is achieved by executing a sleep command (available in the Java programming language) [24], which forces the system to suspend the execution of the process for a specified amount of time [24]. As part of this requirement, a “Latency Milliseconds Delay” property is exposed by the tool, which allows the user to control the number of milliseconds the latency will last (all properties exposed by the tool can be found at Appendix II.).

The failing request attack, on the other hand, simulates an unsuccessful request by returning an empty response before the actual target system is able to act. This is achieved with the techniques described in section 5.3.3.3.1.4.

5.3.3.2.2 *Creating High Memory and CPU usage*

The attacks that create high memory and CPU usage can be further classified as resource attacks. They focus on validating the target system’s resilience to resource scarcity due to other processes’ high consumption.

In order to achieve this parallel use of resources, the attacks leverage a programming technique in Java called threads. A thread is an execution entity inside a program [25]. Threads allow parts of the program to be run concurrently [25]. By running the attacks in parallel to the target system’s main process, its resilience to the degradation of resources over time can be assessed. Each of the attack’s scope and capabilities is specified below.

Memory: The memory attack attempts to fill up the available memory reserved for the Java process. It does so by gradually allocating a block of memory in the form of a byte array. The tool calculates an amount of memory to eat based on the process’s free memory. The byte arrays are then linked to an object kept in memory, effectively causing a memory overhead.

CPU: The CPU attack works by generating a high CPU thread. It does so by keeping the Java process busy with a high computing operation in an iterating loop. As part of this requirement, an additional feature is available to the user: the capability to control the number of times this operation is performed. This property is called CPU Attack Factor and can be configured inside the tool. This feature can, therefore, regulate the amount of stress put into the target system's CPU, as a higher factor equates to a longer exposure of the computationally expensive operation.

5.3.3.2.3 *Changing the contents of the HTTP messages*

This attack works by modifying the content of the body of incoming HTTP requests. The body in these types of requests is represented as JSON [26]. JSON is a human-readable format consisting of key-value pairs and array data types [26].

There are three different variations of this attack: adding new fields, modifying existing fields, or removing fields.

In order to implement this attack, a programming concept in Spring Boot called Filters [27] is used. A filter allows to dynamically intercept requests to transform or use the information contained in them [27]. Custom filters are created in order to intercept the key-value pairs of the request body and modify them. Each filter tackles a particular type of attack:

Adding a new field: This filter takes the request body and appends a new key-value pair. For example, a JSON body for a request before the attack would look like this:

```
{
  "id" : 1,
  "name" : "Chaos"
}
```

And after adding the extra key-value pair would look like this:

```
{
  "id" : 1,
  "name" : "Chaos",
  "chaos_extra_key" : "chaos_extra_value"
}
```

Modifying existing fields: This filter randomly modifies the values in the key-value pairs of the HTTP body. Once a field is selected for modification, the value is modified based on its content. This is achieved through the use of regular expressions, a programming technique that allows defining search patterns [28]. This means that if the value follows a numerical pattern the modified value will be changed to also follow a numerical – but probably unexpected by the target system – value. Other supported patterns include GUID (Globally Unique Identifier), string, array and object field types.

For example, for a field with a numerical value: 4592, the attack will attempt to inject one of the following values: 0, 0.0, -1, 4e10. This modified value is randomly selected from the list.

On the other hand, if the type of the value detected is array or object, the tool will replace it with an empty one, e.g.: {} or [].

An example of this type of attack can be seen below.

JSON body before:

```
{
  "id" : "1",
  "name" : "Chaos",
  "array" : [{
    "array-key" : "array-value"
  }],
  "object" : {"object-key": "object-value"}
}
```

JSON body after:

```
{
  "id" : "12,345.67",
  "name" : "±@#$$%^&*()_",
  "array" : [],
  "object": {}
}
```

Removing existing fields: This filter randomly selects a key from the request body and deletes the key-value pair. The returned JSON body is the same as the original request except for the one missing field.

JSON body before:

```
{
  "id" : 1,
  "name" : "Chaos"
}
```

JSON body after removing the 'name' key-value pair:

```
{
  "id" : 1
}
```

5.3.3.3 Requirement 3: The tool can target different points of message passing interaction

In order to fulfill the requirement of injecting failure at the public method calls the annotations provided by Spring Boot were used to identify the classes where those message passing interactions take place. Out of all annotations provided by Spring Boot, three of them were selected based on the following criteria. The `@RestController` annotation indicates an HTTP endpoint, the `@Service` is used to indicate common functionalities used by the application, and finally, the `@Repository` annotation implies a class that communicates with database resources.

To inject chaos into the workflow of these three targets, the tool makes use of Aspect Oriented Programming (AOP). Besides fulfilling the third requirement this technique also meets the first requirement of minimal instrumentation by allowing to intervene in the normal execution flow without modifying the source code of the target system. This concept is described and introduced in the next subsection.

5.3.3.3.1 *Aspect Oriented Programming*

As defined by the official Spring Framework documentation [29], AOP complements the concepts of Object-Oriented Programming (OOP). In OOP the units for modularity are the classes, however, in AOP the key is the aspect. As the Spring framework documentation describes it, “aspects enable the modularization of concerns such as transaction management that cut across multiple types and objects” [29].

AOP allows to crosscut the workflow of the host application and inject an “advice” at predefined points of the execution, as well as intervene in the method’s actions and modify its arguments.

In the following subsections, some key terms related to AOP are presented. It is important to mention that AOP is implemented for many different programming languages such as C#, C, C++, but here the focus is on the Java and Spring-based solution.

5.3.3.3.1.1 Aspect

An aspect “cuts” across multiple classes and objects. In Java, it means that at the bytecode level the workflow of the program is being aborted and the functionalities of the aspect are being injected by the bytecode weaver [29]. Some examples of these crosscutting concerns given by [29] are collecting statistical metrics, security management, or logging. In a Spring application, an aspect is a regular Java class annotated with `@Aspect`.

5.3.3.3.1.2 Join Point

A *join point* in AOP is the point where the aspect cuts the execution of the program with byte code weaving. It can be an execution of a method or a point of exception handling [29].

5.3.3.3.1.3 Pointcut

The *pointcut* is the predicate that describes the *join point* [29]. It is an expression that can be matched with every desired point to cut the workflow of the program. As an example, a *pointcut* can be created for all public methods [29]:

```
@Pointcut("execution(public * *(..))")  
private void anyPublicOperation() {}
```

The `@Pointcut` annotation defines the method as a *pointcut*, and the predicate matches the execution of every single public method with the usage of regular expressions.

5.3.3.3.1.4 Advice

Advices are the code snippets that get executed – injected into the bytecode – when the program meets a *join point* [29]. The following types of *advices* are used in the tool:

- *Before*: the advice gets executed before the pointcut. The advice does not have any means of stopping or intervening the flow of the program [29].
- *Around*: the advice performs the action before and after the execution of the *pointcut* having the ability to stop the workflow of the *join point* [29].

5.3.3.4 Requirement 4: Attacks can be targeted using a probability property

If chaos is enabled in the tool, then the user can set a probability property to manage the frequency of attacks. If it is set to 1, any activated perturbation is guaranteed to occur. Meanwhile, when the probability is set to 5, there will be a 1 in 5 chance that the perturbation will be injected. This is achieved by picking a random number between 1 and 5 and comparing it against 5, meaning a 20 percent chance of a match. This property can be adjusted with the same endpoint as attack activation: “<host-system-base-url>/chaos-plugin/attacks”. By sending a JSON formatted POST message to the endpoint the frequency will be changed. For instance:

```
{
  "attackFrequency" : "5"
}
```

5.3.3.5 Requirement 5: The tool provides a functionality to stop running attacks

The fifth requirement states the need for a functionality to stop all running experiments in order to restore the normal and steady-state of the target system. Given any breakdown of the system caused by the tool, further damages can be prevented by using this safety feature.

The solution is based on RESTful endpoints exposed by the chaos engineering tool and a properties class that can be changed via the endpoints. These are explained in the next subsections.

5.3.3.5.1 *Status of Attacks*

Every attack that can be executed by the tool has a corresponding property (shown in Appendix II) which keeps track of its activated or deactivated status. These properties are Boolean values contained by the aforementioned properties class. Each property can be changed via the endpoint exposed at “<host-system-base-url>/chaos-plugin/attacks” with an HTTP POST message. The POST body should contain the properties to be changed in a “key”: “value” manner, using JSON format. For example:

```
{
  "latencyAttackActivated" : "true",
  "failRequestAttackActivated" : "false"
}
```

5.3.3.5.2 *Enabling the Chaos Tool*

Executing an HTTP POST message (with an empty body) to the path “<host-system-base-url>/chaos-plugin/enable” results in the enabled state of the tool. This means that specific chaos attacks having an activated status can be executed starting from this moment.

5.3.3.5.3 *Disabling the Chaos Tool*

A POST message executed to the path “<host-system-base-url>/chaos-plugin/disable” disables the tool and sets the status of each attack to deactivated. Some of the implemented chaos attacks are designed to run for a few seconds. The latency attack should last for the milliseconds defined by the user, and the attacks for heightened memory and CPU usage are running as long as they fulfill their purpose of eating up memory or CPU resources.

Therefore, a proper solution was implemented to be able to stop these long-running attacks at any desired point. Since these attacks run in loops, the solution is a recurring inspection of the enabled state of the tool during the execution. If the tool gets disabled, these attacks are halted.

5.3.3.6 Requirement 6: The tool provides standard output logging around chaos events

While conducting tests and experiments the tool offers a feature to observe the on-going chaos operations. This is implemented by adding logging features in a manner that fits the target system's logging mechanisms. Since the architectural approach of the tool is based on the concept of Spring Boot Starters (elaborated at chapter 5.3.1) it is unavoidable that any output produced to the standard output is mixed with the target system's outputs.

Therefore, while keeping the common logging platform with the target system, for comprehensibility and readability reasons an additional solution is implemented. The tool exposes an endpoint at “<host-system-base-url>/chaos-plugin/log” where only the logs of the chaos engineering operations can be observed. Each record contains an exact timestamp, the name of the attack producing the record, and information regarding the current activity of the attack.

However, for the endpoint to expose this data, the logs (besides written on disk) are constantly kept in the memory by the tool – implying a fast-growing object. To handle this growing object in memory, the tool has an additional management endpoint at “<host-system-base-url>/chaos-plugin/log/flush”, that instantly discards the existing log and assigns a new one. To avoid the efforts of doing this manually every time, the log object is also reinitialized after 1000 lines of content.

5.4 Activity 4: Demonstration

In this activity, the usage of the developed chaos engineering tool is demonstrated through experiments. The design of the experiments is based on the four steps described in section 5.1.1. First, a metric is chosen to determine the normal behavior of the system [10]. Second, a hypothesis about the target system's resilience is formulated which can either be validated or falsified [10]. Third, the experiment is run, and finally, the hypothesis is evaluated so that conclusions can be drawn about the resilience of the system [10].

In the following subsections, the environment and device used for the experiments are described. Furthermore, in Table 2 the conducted experiments are detailed along with the formulated hypotheses, different scenarios, and their evaluations.

5.4.1 Environment

In order to conduct the experiments, the target system's local and staging environments were used. The local environment is the one used by developers to test their features locally. This environment does not support real interactions with the MNOs. This means that no real messages can be sent to the cellphones using this setting. Still, developers are able to run their tests in this environment by using simulators. These simulators act like servers running locally and returning default responses as if they are issued by the MNO. The developers can then verify that the internal logic is executing as expected.

The staging environment, on the other hand, exhibits the same functionality as the production environment where requests can be forwarded to MNOs and their callback responses can be received. This environment is capable of sending real messages to cellphones to test the message receiving functionality. Furthermore, the staging environment provides a replica of the production environment where among other settings dual-core processors, proxies, SSL termination, and horizontal scaling of servers are used.

Naturally, there is a production environment, but as a first implementation phase and following the recommendations on how to start chaos engineering (section 5.1.1), it was agreed by the company that experiments would only be conducted in the previously mentioned environments. This is due to the potential risks and impacts that running in production could carry, as mentioned by practitioners [16][17].

5.4.2 Device

In order to conduct experiments in the staging environment, a cellphone device is needed. The device needs to be able to receive the specific kind of text messages that the target system is sending. Therefore, the requirements for this mobile phone are:

- Android operating system with a version higher than 5.0
- SIM card issued by an MNO that is supported by the target system
- Android Messages application installed

5.4.3 Metric

As described in section 5.1.1 on how to do chaos engineering, in order to evaluate the system's behavior a metric has to be defined. This can be a combination of *operation* and *user experience* metrics. For these experiments, three metrics were defined.

1. From a *user experience* point of view, whether or not the message is successfully received on the phone.

From the *operational* perspective, the successful handling of errors in the application side which includes:

2. Proper exception handling logic. This means no unhandled exceptions or exception stack traces in the logs.
3. Proper error messages. This means that the end-user is correctly notified when something unexpected happens.

5.4.4 Description of the experiments

In this subsection, the conducted experiments are summarized. All the attacks implemented by the chaos tool are executed against the target system in the local and staging environments. Table 2 summarizes the results, each row describing an individual experiment. The columns of the table can be interpreted as follows:

- **EXPERIMENT NO:** the ID of the experiment
- **EXPERIMENT NAME:** the title of the attack
- **DESCRIPTION:** short description of the experiment
- **HYPOTHESIS:** the formulated hypothesis
- **SN NO.:** the ID of the scenario within an attack
- **METRIC EVALUATION:** indicates the evaluation of the following metrics:
 1. Whether the message was successfully received on the device
 2. Whether the target system has proper error handling logic
 3. Whether appropriate error messages were issued to the caller
- **HYPOTHESIS EVALUATION:** the hypothesis is validated or falsified
- **COMMENT:** a short summary of the findings associated with the experiment

The detailed description of each of the experiment scenarios can be found in Appendix III.

Table 2: Experiments Description

EXP NO.	EXPERIMENT NAME	DESCRIPTION	HYPOTHESIS	SN NO.	METRIC EVALUATION			HYPOTHESIS EVALUATION	COMMENT
					1	2	3		
1	Introducing Latency	This experiment tests the target system's resilience to artificially added latency.	Target system is resilient to artificially introduced latency	1	✓	N.A.*	N.A.	VALIDATED	Asynchronous message processing capabilities made the target system resilient to the introduced latency and caused no timeout issues.
2	Modifying Fields	These experiments aim to test the resilience of the target system at receiving modified fields in the HTTP POST body.	Target system is resilient to faulty values in requests	1	✗	✓	✗	FALSIFIED	The target system responded with an error message that did not reflect the problem of the request.
3				✗	✗	✗	FALSIFIED	An unhandled <i>NullPointerException</i> caused the target system to respond with HTTP 500.	
4				N.A.	✓	✓	VALIDATED	The incorrect value was handled by sending an appropriate error message highlighting to the end-user the wrong value.	

5	Removing Fields	These experiments test the target system's resilience to missing JSON fields in HTTP POST requests.	Target system is resilient to missing fields in requests	1	✓	✓	✗	FALSIFIED	The error was silenced by the target system, the workflow continued without disturbance, failing to notify the caller of flaws in the request.
6				2	✗	✓	✗	FALSIFIED	After an exception, the target system responded with a misleading error message.
7				3	✗	✗	✗	FALSIFIED	An unexpected use case of consecutive calls led to an unhandled exception.
8				4	N.A.	✓	✓	VALIDATED	The target system reacted with appropriate error messages and notified the caller of violated requirements.
9				5	N.A.	✓	✗	FALSIFIED	The target system ignored the erroneous request and responded with HTTP 200.
10				6	N.A.	✗	✗	FALSIFIED	A <i>NullPointerException</i> caused the target system to return HTTP 500 with no proper error message.
11				7	N.A.	✗	✗	FALSIFIED	A <i>NullPointerException</i> caused the target system to return HTTP 500 with no proper error message.

12	Adding Fields	These experiments are designed to test the failure handling capabilities of the target system to HTTP requests containing extra fields in the POST body.	Target system is resilient to added extra fields in requests	1	✓	N.A.	N.A.	VALIDATED	The request was successfully handled, and no error messages were found in the target system's logs.
13				2	✓	N.A.	N.A.	VALIDATED	The request was successfully handled, and no error messages were found in the target system's logs.
14	Failing Requests	These experiments test the target system's resilience to failed requests.	Target system is resilient to failed requests	1	✓	✗	✓	FALSIFIED	The failed request led to an error in the target system that logged personally identifiable information (PII), leading to GDPR non-compliance.
15				2	✓	N.A.	N.A.	VALIDATED	Asynchronous message processing capabilities made the target system resilient and caused no issues.

16	Creating High Memory Usage	This experiment tests whether the target system is resilient to high memory consumption scenarios.	Target system is resilient to high memory usage	1	✓	N.A.	N.A.	VALIDATED	Despite the high memory usage, the request was properly handled, and a successful response was received.
17	Creating High CPU Usage	This experiment tests the resilience of the target system to high CPU usage.	Target system is resilient to high CPU usage	1	✓	N.A.	N.A.	VALIDATED	As the CPU usage grew, so did the response times, taking almost one minute and a half, yet the target system was able to gracefully handle the request.
18	Stop all running attacks	These experiments test the tool's ability to stop all running attacks when the chaos tool is disabled.	The tool is able to stop running attacks	1	N.A.	N.A.	N.A.	VALIDATED	No new attacks were introduced as a result of disabling the tool.
19				2	N.A.	N.A.	N.A.	VALIDATED	After disabling the tool, all attacks running on threads were halted.

*N.A.: (Not Applicable) This means that the metric is not applicable to the scenario.

5.5 Activity 5: Evaluation

In this activity, the artifact created in the previous steps is evaluated [15]. This means that its effectiveness is analyzed. The objectives from activity 2 are compared to the solutions described in activity 3, and the artifact's design and architecture are evaluated. The following subsections describe how each objective defined in section 5.2 was fulfilled and achieved.

5.5.1 Objective 1: Minimal instrumentation of the target system

The first objective aimed to assure that the target system needed minimal instrumentation to host the chaos tool. Practitioners have stated the benefits of being oblivious to the target system's internals [1][14]. By creating a Spring Boot Starter application and following the man-in-the-middle approach discussed in 5.3.3.1.1, the tool's first objective was accomplished. Furthermore, by using similar technologies as the host application, functionalities that are specific to the Java and Spring technology stacks could be exploited.

Another advantage of this solution is that the developer effort needed to use the tool is reduced. The developer does not need to make several modifications to the existing code base in order to make the tool work. The end-user would only need to install the plugin and start using it to be able to see the first results.

On the other hand, this approach manifests a disadvantage: the tool has constraints in terms of technical solutions introduced by the pre-defined technology stack.

5.5.2 Objective 2: The tool introduces chaos at the message passing level

The second objective defined the scope of the perturbations introduced by the chaos tool. In the first activity of the methodology (section 5.1.3), examples could be found in the literature for different approaches on intercepting network traffic with chaos experiments [1] [30] by injecting artificial latency, failing requests or maxing out CPU and memory resources.

The presented solution is different as it introduces chaos at the message passing level by also accessing the contents of messages and not just packets on the network. This is a novelty factor in this thesis.

By implementing REST Interceptors (described in section 5.3.3.2.1), the chaos tool can capture the arrival and the end of the processing of requests. This way artificial latency can be introduced to the message exchanges or the request can be completely discarded before the target system gets access to it. Introducing latency is a feature that was mentioned in previous research by Basiri et al [10].

The Filters described in section 5.3.3.2.3 help to intercept incoming requests and give the tool the ability to change them. A limitation of this solution is that only JSON POST body requests were considered; therefore, the tool would need to be modified in order to work with XML based contents.

The thread-based resource exhausting attacks in the chaos tool tackle a different area. As mentioned by Rosenthal et al [17], when creating chaos experiments, practitioners usually do not think about resource issues, only software-related ones. Therefore, this tool provides a novelty factor in the sense that it offers software attacks on hardware resources.

The usage of AOP described in section 5.3.3.3.1 allows to implement cross-cutting functionalities – i.e. inject the tool’s logics into the workflow of the target system. This way the chaos tool could intervene in many different processes of the system. The pointcuts were defined to match all Java classes using Spring’s `@RestController`, `@Repository`, and `@Service` annotations. It is also possible to declare pointcuts for specific packages or even method names of the target system [29]. This is a possibility not explored by the tool that can be further investigated.

5.5.3 Objective 3: Attacks can be targeted with a variable probability

The third objective aims to address a desired functionality that is mentioned in the literature by many researchers [1][3][6]. Giving a random factor to the execution of the attacks can result in discovering unexpected use cases, whereas directing the attack with probability factor 1 the target system’s specific error handling capabilities can be tested. The presented chaos tool enables the user to execute attacks with this level of control.

5.5.4 Objective 4: The tool can be stopped at any time

This objective was claimed as a must-have by practitioners in the literature [31]. The ability to stop any running attacks at any time was implemented in a simple manner that only needs an HTTP POST request to stabilize the system. As Hornsby at [31] says, any chaos engineering tool must have an “emergency stop button” to halt all the chaos at once. Moreover, the principles of chaos highlight the importance of minimizing the blast radius in order to prevent any unwanted breakdown of the system.

5.5.5 Objective 5: The tool grants observability of the scope of the chaos

The fifth objective aims to grant observability of the chaos tool’s operations. Will et al [16] mention that providing monitoring for the chaos is essential in order to understand how the system behaved before, during, and after the experiment.

This functionality was implemented in two approaches:

- The tool shares the standard output with the target system - therefore, its logs can be found there. However, the combined logging affects the readability and comprehensibility of the chaos logs, hence an additional solution was defined as described in section 5.3.3.6.
- The logs of the chaos tool are collected and exposed in an endpoint. This solution is arguably lacking the ability to store older logs since the endpoint flushes the data after 1000 lines.

By comparing the presented tool to the related work section an additional observation can be made. The Chaos Orca project presented in 3.4 implements a monitor component that analyzes the impact of the surrounding physical resources. While there is no mention of the need for such components in the literature review, this is a feature that the tool’s design did not take into consideration.

5.6 Activity 6: Communication

As part of the communication activity, this thesis will be made publicly available at Malmö University's Electronic Publishing platform (MUEP). Moreover, the project results are presented to the community, by defending the thesis at this University and showcasing the tool at the collaborating company.

The results of this chaos engineering tool, as well as its findings, mark a contribution to the existing body of knowledge. This thesis is a stepping-stone for future researchers and practitioners to continue exploring in a field where there is still room for research to be done.

6 Discussion and Conclusion

In this chapter, the answers to the research questions are presented and discussed. Furthermore, the contributions of this thesis to chaos engineering and the software testing area are elaborated. Finally, the threats to validity are analyzed.

6.1 Answering Research Questions

In this section, each research question of this thesis is answered.

6.1.1 RQ1: What is a reasonable methodological approach to develop a chaos engineering tool?

In order to conduct this research Peffers et al Design Science for Research Methodology in Information System (DSRM) was used. The methodology's activities served as a guide that laid out the necessary steps to develop the chaos engineering tool.

In the first activity, a literature review was conducted. This step allowed to discover the existing knowledge on how to start a chaos engineering project as well as its value and relevance. The second and third activities of the methodology focused on defining the objectives as well as functional requirements for the chaos tool to be developed. Additionally, in the third activity, the design and architecture of a tool that could meet those demands were presented. Finally, the demonstration and evaluation activities allowed assessing the finalized tool against the previously defined objectives.

As an outcome of following this methodology, the main artifact of this research came to life: a novel chaos tool. Moreover, its results proved to be of value in assessing the target's system resilience. Peffer's methodology demonstrates its merit by naturally allowing to develop and evaluate the effectiveness of such a tool. Hence Design Science for Research Methodology proved to be a reasonable and methodological approach for practicing chaos engineering.

6.1.2 RQ2: What are the functional requirements of a chaos engineering tool for the target system?

Based on the first activity of Peffer's methodology, the researchers' and practitioners' learnings on previous chaos engineering efforts were discovered. These learnings were

used to define the tool's main objectives. For instance, the principles of chaos stated the importance of varying real-world events and minimizing the blast radius, two of the objectives of the tool. As a result of this, a total of five objectives were identified (section 5.2), which served as the foundations for the development of the tool.

In the third activity, six functional requirements were further defined and scoped (section 5.3.2). These requirements aimed to fulfill the main objectives that were previously laid out. The requirements are an essential part of the process, as they allow to outline the functionality and behavior of the tool.

By specifying and defining the functional requirements of the tool, the second research question was answered.

6.1.3 RQ3: How can a chaos engineering tool be designed and architected for the target system?

In order to answer this research question, during the third activity of the research methodology (section 5.3.3), the tool's architecture and the technical aspects of the design were described. The design's goal is to fulfill the requirements, as such the "man-in-the-middle" concept was introduced. It allows injecting an actor in the communication between two entities. This principle in connection with the technical solution of Spring Boot Starter, allowed the architecture to comply with the requirement of minimal instrumentation of the target system.

Aspect Oriented Programming, a programming technique that helps to create cross-cutting solutions, allowed to fulfill the message passing level requirement. Moreover, using this solution specific points of the application could be targeted at achieving the third requirement. By exposing configurable properties, the user of the tool is able to modify the probability of the attacks. The tool also provides an interface to query the attack's logs as well as a functionality to stop all running attacks, helping satisfy the fourth, fifth, and sixth requirements.

These architectural decisions ensure the compatibility of the chaos tool with other Java-based applications. Hence, these findings could be validated against other systems with similar characteristics.

All of these technical implementations and the technology stack of the target system shaped the design and architecture of the chaos tool, answering the third research question.

6.1.4 RQ4: What are some of the attributes that make a system resilient and how can chaos engineering be used to build confidence on them?

The goal of this research was to build an understanding of the factors that make Chaos Engineering a suitable option for testing a system's resilience. By designing a Chaos Engineering tool and evaluating it, relevant data was collected that allows analyzing some of the characteristics that make systems fault-tolerant.

Following the chaos features defined in requirement 2 (section 5.3.3.2), the resilience capabilities of the target system were verified. Hence the first trait to be evaluated was the ability of systems to handle latency in the requests. The target system's asynchronous request handling capabilities showed its ability to deal with delays in the messages. This mechanism does not block the execution waiting for a response, instead, it provides a callback that clients can tap into. The chaos tool was effective at proving the value of this target system's attribute hence building confidence in its resilience mechanisms.

Second, regarding the system's capability to handle failed requests, a relevant result was discovered. The chaos engineering tool helped the system to enter a control flow where the user's phone number is logged. However, this phone number was displayed in full form, overriding the target system's censor mechanism that obscures such numbers to comply with GDPR [32]. While this metric was not considered when designing the experiments, it proved to be a very valuable finding. These types of bugs must be fixed to avoid any legal implications. Failed requests proved to be a type of failure that systems must be able to handle as part of their error handling mechanisms.

Third, concerning the resilience of systems to high resource consumption, the chaos tool was proven effective at generating high memory and CPU load. This was evident from the data collected by the monitoring tools and logs. Moreover, the high CPU usage showed to have an effect on the system responses' times, from just a few milliseconds to almost a minute and a half. Having said that, the target system was still capable of handling requests by sending and receiving successful responses. These experiments exposed the resource

management attributes of the system and how this chaos tool can be used to build confidence in their resilience.

Finally, moving to the modification of the HTTP request body, different types of perturbations were explored. For example the addition, modification, and removal of the fields. In this regard, an attribute that makes systems resilient was found: its ability to validate the content of messages. Through the use of the tool different *NullPointerExceptions* were found in the code resulting in HTTP 500 responses, indicating an internal server error. Moreover, the way the system handled end-user errors exhibited some inconsistencies. While some error responses highlighted the exact incorrect field, others simply returned empty responses, generic error messages, or misleading ones. These findings show that resilience to erroneous requests is necessary to build confidence in the fault-tolerance assumptions of systems.

6.2 Contribution to Chaos Engineering

The field of chaos engineering is still new and emerging, hence there is a need for research and new perspectives on the topic. This thesis contributes with a novel approach to designing and architecting a chaos engineering tool. It implements features based on the learnings of previous practitioners and confirms the need for utilities in a chaos engineering tool such as observability over the operations, configurable properties, and an “emergency stop button” [31]. By conducting experiments it was validated that the principles of chaos [18] are reasonable steps to follow while practicing chaos engineering. This thesis also identified multiple attributes that software systems should have in order to be considered resilient.

6.3 Contribution to Software Testing

This thesis contributes to the software testing field by implementing a tool that facilitates resilience testing. This tool was able to shed light on the failure handling weaknesses of a software system. In a matter of only a couple of months and the working hours of two engineers, it was possible to build a tool that greatly reduces the otherwise manual testing labor and allows to find potentially costly failures. These results prove the benefits that many companies could experience in their return of investment by adopting this practice.

6.4 Threats to validity

This section aims to identify some threats to validity regarding the thesis.

As for internal validity, there is a possible bias in the selection of papers included in the related work. Due to the field of chaos engineering being new, an exhaustive list of all relevant publications in the topic may not have been collected. Best practices mostly emerged from the industrial innovation and not much empirical research has been conducted.

Additionally in regards to internal validity, there is a threat introduced by the conducted experiments. A greater quantity or different scenarios could have yielded some new conclusions and results.

An external validity threat is the generalizability of the results. Experiments were run against one scalable distributed application. Furthermore, the requirements, design, and architecture of the tool were biased by the knowledge gathered about this system. However, the results could still be generalized to other Java-based systems with message passing interactions.

7 Future Work

The chaos engineering tool presented in this thesis can be extended in several ways. A few ideas are listed here.

7.1 Supporting other message passing channels and content types

While the tool's attacks cover the HTTP channel of the targeted system, it only does so when the body is JSON formatted content. Therefore, a solution for XML support can be implemented. It is important to mention that HTTP is not the only interaction channel in software systems. For example, a feature could be developed to intercept gRPC (gRPC Remote Procedure Calls) communications.

7.2 Improve user control over attacks

The chaos tool can be configured in terms of targets and activated attacks. However, some of those attacks have an uncontrollable factor: the user of the tool cannot decide which field should be removed or modified. Therefore, a solution can be implemented to enable these specific settings. Furthermore, the targets can be expanded if the user wants to intercept a specific package, class, or method. These features would improve control over the chaos tool.

7.3 Automation

In its current form, the chaos tool needs manual interaction to start operations, and user settings to specify the targets and attacks of interest. A possible future work here is implementing a user interface that automatically turns on the chaos tool and enables random attacks on randomly chosen targets. This way unexpected failures can be injected into the target system, unraveling latent bugs.

Another possibility of user-friendly automation is the development of a built-in notification feature that alerts the user when a bug is found in the target system. This would enable the user to run the attacks for longer time periods and only check the outputs if there is an error notification.

Appendix I: Endpoint mapping

This appendix shows a mapping of endpoints and functionalities exposed by the developed chaos tool. All the endpoint's paths are relative to the host application.

ENDPOINT	GET	POST
/chaos-plugin/attacks	Returns the list of attacks	Changes attacks' properties
/chaos-plugin/targets	Returns the list of targets	Changes targets' properties
/chaos-plugin/enable	-	Enables the tool
/chaos-plugin/disable	-	Disables the tool
/chaos-plugin/log	Returns the logs of the tool	-
/chaos-plugin/log/flush	Flushes the logs	-

Appendix II: Chaos Tool Properties

This appendix shows a table of all the properties and the functionalities that can be customized in the chaos tool.

PROPERTY NAME	FUNCIONALITY	VALUE
attackFrequency	Probability factor for the attacks (section 5.3.2.4)	[1, Integer.MAX_VALUE] *
latencyActivated	Enables or disables the latency attack (section 5.3.2.2.1)	[true false]
latencyMillisecondsDelay	Specifies the introduced latency in milliseconds	[0, Long.MAX_VALUE] **
failRequestActivated	Enables or disables the failing request attack (section 5.3.2.2.1)	[true false]
cpuActivated	Enables or disables the CPU attack (section 5.3.2.2.2)	[true false]
cpuAttackFactor	Specifies the number of loops to attack the CPU (section 5.3.2.2.2)	[1, Integer.MAX_VALUE]
memoryActivated	Enables or disables the memory attack (section 5.3.2.2.2)	[true false]
addFieldActivated	Enables or disables the add field attack (section 5.3.2.2.3)	[true false]
modifyFieldsActivated	Enables or disables the modify field attack (section 5.3.2.2.3)	[true false]
removeFieldActivated	Enables or disables the remove field attack (section 5.3.2.2.3)	[true false]
serviceTargetActivated	Enables or disables the @Service class targets	[true false]
controllerTargetActivated	Enables or disables the @RestController class targets	[true false]

repositoryTargetActivated	Enables or disables the @Repository class targets	[true false]
----------------------------------	--	--------------

* The maximum value for an Integer in Java is 2 147 483 647.

** The maximum value for a Long in Java is 9 223 372 036 854 775 807.

Appendix III: Experiments Description

This appendix is a detailed description of the conducted experiments from section 5.4.4.

1. Introducing Latency Experiment

The target system's resilience to artificially added latency is tested in these experiments. The goal is to explore if delays can be tolerated by the message processing parts of the target system.

1.1 Hypothesis

Target system is resilient to artificially introduced latency.

1.2 Experiment Design

For the introducing latency attack experiment, the chaos tool needs to be configured properly. The endpoints of the tool are used to enable and activate the attack:

```
{
  "latencyActivated" : true,
  "attackFrequency" : 1,
  "isChaosPluginEnabled" : true
}
```

The target of the attack is set to the REST endpoints:

```
{
  "controllerTargetActivated" : true
}
```

The configuration of the introduced latency in milliseconds is shown in the scenarios below.

1.2.1 Scenario 1

The experiments considered many different settings for adjusting the amount of latency. This can be done using the endpoints of the chaos tool. For example:

```
{
  "latencyMillisecondsDealy" : 3000
}
```

```
{  
  "latencyMillisecondsDealy" : 15000  
}
```

```
{  
  "latencyMillisecondsDealy" : 30000  
}
```

For the experiments, many different requests to the target system were made.

1.2.1.1 Results of the experiment

In the chaos tool's logs the introduced latencies can be observed:

```
LatencyAttack Starting...  
LatencyAttack Intercepted Path: /<MNO1>/.../.../messages  
  
LatencyAttack Starting...  
LatencyAttack Intercepted Path: /<MNO2>  
  
LatencyAttack Starting...  
LatencyAttack Intercepted Path: /<MNO3>/.../message_callback
```

1.2.1.2 Evaluate the Hypothesis

Through conducting the experiments, it can be observed that the target system was tolerant to the attacks. Its asynchronous message processing capabilities made it resilient to the introduced latency and caused no timeout issues nor did it alter any metrics.

After running a few experiments that introduced latency in the requests, the hypothesis was validated.

2. Modify Field Experiment

These experiments aim to test the resilience of the target system at receiving modified fields in the HTTP POST body. The system has no control over the contents of the messages that clients and MNOs send. Therefore, it is the target's responsibility to validate bad formatted data.

2.1 Hypothesis

Target system is resilient to faulty values in requests.

2.2 Experiment Design

For this experiment the modify fields attack is activated and targeted to the controller. The chaos plugin configuration endpoints are used to make the attacks configuration as follows (other fields set as false):

```
{
  "modifyFieldsActivated" : true,
  "attackFrequency" : 1,
  "isChaosPluginEnabled" : true
}
```

And the target configuration as this (other fields set as false):

```
{
  "controllerTargetActivated" : true
}
```

2.2.1 Scenario 1

For the first scenario the attack has been targeted to a request that expects the following valid schema:

```
{
  "to" : "+36302*****",
  "message" : {...}
}
```

2.2.1.1 Results of the experiment

After executing the attack and inspecting the chaos tool logs the output shows that the "to" field has been changed as follows:

```
ModifyFieldsAttack Original field was: "to":"+36302*****", new field is: "to":"12,345.67"
```

And the error message the system returns back to the client is:

```
{
  "ref_id" : "...",
  "error" : "Could not handle the input",
  "error_code" : 1010,
  "field_errors" : [
    {
      "field" : "root",
      "errors" : [
        "The body must be a valid JSON"
      ]
    }
  ]
}
```

2.2.1.2 Evaluate the Hypothesis

As evidenced by the returned response, the system is not able to handle the error properly. The output shows that there is something wrong in the request, however, the error message does not reflect the real problem. The message – despite its modification – is still a valid JSON. A proper response should have highlighted that the “to” field is not a valid phone number. Therefore, the hypothesis was falsified in this scenario.

2.2.2 Scenario 2

In this scenario the system expects a particular request from the MNO (known as a callback) to have the following schema:

```
{
  ...,
  "message" : {<message-content>}
}
```

2.2.2.1 Results of the experiment

After the attack has been executed it can be inspected in the logs that the “message” field has been changed as follows:

```
ModifyFieldsAttack Original field was: "message":"<message-content>", new field is: "message": "{}"
```

Furthermore, the system’s logs show the following results:

```
ExceptionHandlerExceptionResolver - Resolved  
[java.lang.NullPointerException]  
ERROR status: 500 request body: {"message":{},..} response body null
```

2.2.2.2 *Evaluate the Hypothesis*

The target system is not handling the request as expected. The end result is a 500 response, meaning that there was an internal server error. Indeed, as seen by the logs, there was a *NullPointerException* raised in the code. In this scenario the system should be able to properly handle the exception and return an appropriate error message, hence the hypothesis was falsified.

2.2.3 Scenario 3

For this scenario the system expects a request that contains a field “message_id” which should be a valid GUID:

```
{  
  ...,  
  "message_id" : "<GUID>"  
}
```

2.2.3.1 *Results of the experiment*

After executing the attack, the logs show that the following changes were made to the request:

```
ModifyFieldsAttack Original field was: " message_id ":"<GUID>", new  
field is: " message_id ":"a-cat"
```

The target system returns a response that looks like this:

```
{  
  "ref_id" : "...",  
  "error" : "Not a valid property value",  
  "error_code" : 1005,  
  "field_errors" : [  
    {  
      "field" : "root",  
      "errors" : [  
        "'a-cat' is not a valid value for property"  
      ]  
    }  
  ]  
}
```

2.2.3.2 *Evaluate the Hypothesis*

As shown by the request's response the target system was able to properly handle the incorrect value by sending an appropriate error message highlighting to the end user the wrong value. Therefore, the hypothesis was validated in this scenario.

3. Remove Field Experiment

The experiments described in this section are centered around testing the target system's resilience to missing JSON fields in HTTP POST requests. Missing data from requests can either be ignored or the target system can make efforts to inform the caller about the problem.

3.1 Hypothesis

Target system is resilient to missing fields in requests.

3.2 Experiment Design

In order to conduct experiments with the remove field attack, the chaos tool needs to be configured. The endpoints of the tool are used to enable and activate the attack:

```
{
  "removeFieldsActivated" : true,
  "attackFrequency" : 1,
  "isChaosPluginEnabled" : true
}
```

The target of the attack needs to be set to REST endpoints:

```
{
  "controllerTargetActivated" : true
}
```

3.2.1 Scenario 1

The target system expects a request which under normal circumstances should have the following fields (those fields that are irrelevant for this experiment are omitted):

```
{
  "subscription" : "...",
  "message":{
    ...
    "attributes":{
      ...
      "event_type" : "<event-from-predefined-list>"
    }
  }
}
```

3.2.1.1 Results of the experiment

The Remove Field Attack clears the whole "message" field from the incoming request.

This operation was logged by the chaos tool:

```
RemoveFieldAttack Original request was: {"subscription" : "...",  
"message":{"attributes" : {..., "event_type" : "<event-from-  
predefined-list>"}}, removed field is: "message": {...}
```

3.2.1.2 *Evaluate the Hypothesis*

After running the experiment, by inspecting the logs it can be observed that the “event_type” field was expected to be a value from a predefined list. With the absence of this field (due to being inside the removed “message”) the whole request was categorized as “Unsupported”. However, the response from the target system to this attacked request was a simple HTTP 200 (OK), indicating no error to the caller.

The error caused by the chaos tool was silenced by the target system. It led to the workflow continuing without disturbance, but the target system failed to notify the caller of any potential flaws in the sent request, therefore the hypothesis was falsified.

3.2.2 Scenario 2

In this experiment the target system expects requests to follow the schema shown below (those fields that are irrelevant for this experiment are omitted):

```
{  
  "to" : "...",  
  "event_id" : "...",  
  "event" : {...}  
}
```

The remove field attack was conducted for three cases of this request: the removal of the field “to”, “event_id” and “event”, one at a time.

3.2.2.1 *Results of the experiment*

The results of this operation can be seen in the chaos log shown below:

```
RemoveFieldAttack Original request was {"to" : "...", "event_id" :  
"...", event : {...}}, removed field is: "to": "..."
```

3.2.2.2 *Evaluate the Hypothesis*

The results from the three cases of this scenario are all summarized together because they indicated the same issue of failure handling.

In the target system, the requests are validated by a Spring Boot mechanism. If this validation fails – implying a non-conform value in the request – a specific exception is thrown by the Spring Framework. This exception is not caught by the target system. Instead, the Spring Framework responds with a default HTTP 400 and a message saying “Non-Compliant JSON data”.

The error message is not appropriate, and the response should be formulated by the target system indicating the specific field that was missing. Hence, the hypothesis was falsified.

3.2.3 Scenario 3

In this experiment, two subsequent requests are being attacked by the tool. The first request has the following schema (the irrelevant fields for this experiment are omitted):

```
{
  ...
  "to" : "...",
  "message" : {...}
}
```

The second request contains these fields:

```
{
  "to" : "...",
  "event_id" : "...",
  "event" : {...}
}
```

The experiment considered the removal of “to” field from the first request and the second was left untouched. It is important to mention that these calls are closely related and are logical to be sent in this order to the target system’s API.

3.2.3.1 Results of the experiment

The chaos log below shows the removal of the “to” field from the request:

```
RemoveFieldAttack Original request was {..., "to" : "...", "message" : {...}}, removed field is: "to": "..."
```

3.2.3.2 Evaluate the Hypothesis

The observation of these consecutive calls led to the discovery of an unexpected use case in the target system. The first request was intended to start a conversation on a mobile

device (where there was no interaction before). This request failed due to the missing “to” field. The target system responded with an appropriate error message stating that this field was mandatory. Ignoring the failed first request, the second one was sent, and resulted in an uncaught exception, claiming that “Conversation ID not present”. This sheds light on the problem that the target system did not expect the caller to send the second request when the first one failed.

The target system should be prepared for such erroneous requests where the caller does not consider consequences from previous interactions. Therefore, the hypothesis was falsified.

3.2.4 Scenario 4

In the fourth scenario requests to the target system are expected to have the following fields (the irrelevant fields for this experiment are omitted):

```
{
  ...
  "to" : "...",
  "message" : {...}
}
```

While conducting experiments both the removal of “to” and “message” fields were considered.

3.2.4.1 Results of the experiment

In the logs of the chaos tool the deletion of the fields can be observed:

```
RemoveFieldAttack Original request was {...,"to" : "...","message" : {...}}, removed field is: "to":"..."
```

```
RemoveFieldAttack Original request was {...,"to" : "...","message" : {...}}, removed field is: "message":{...}
```

When the “to” field was deleted from the request the target system responded with an error message stating:

```
{
  "ref_id": "...",
  "error": "Missing required property",
  "error_code": 1004,
  "field_errors": [{
    "field": "message",
    "errors": [
      "property 'to' is required and must not be empty"]
  ]
}
```

In case of “message”, a similar response was received:

```
{
  "ref_id": "...",
  "error": "Missing required property",
  "error_code": 1004,
  "field_errors": [{
    "field": "message",
    "errors": [
      "property 'message' is required and must not be empty"]
  ]
}
```

3.2.4.2 Evaluate the Hypothesis

In both cases the target system was resilient to the faulty requests, validating the hypothesis. It reacted with appropriate error messages and notified the caller of potential violated requirements.

3.2.5 Scenario 5

In this scenario the targets system’s endpoint expects a request following this schema:

```
{
  "accountId" : "...",
  "events" : [...]
}
```

The experiments considered the removal of both “accountId” and the “events” array, one at a time.

3.2.5.1 Results of the experiment

The logs of the chaos tool show the removal of the aforementioned fields:

```
RemoveFieldAttack Original request was {"accountId" : "...", "events" : [...]}, removed field is: "accountId" : "..."
```

```
RemoveFieldAttack Original request was {"accountId" : "...", "events" : [...]}, removed field is: "events" : "[...]"
```

3.2.5.2 Evaluate the Hypothesis

In the case of deleting “accountId” from the request, the target system showed no disturbance and went on to process the message successfully. When “events” was removed, the target system responded with HTTP 200.

However, this might not be the intended reaction since the request was missing its purpose by not having any “events”. Therefore, a proper error message should be issued in this case for the caller. Hence, the hypothesis was falsified.

3.2.6 Scenario 6

In this experiment the target system expects requests following the schema shown below (those fields that are irrelevant for this experiment are omitted):

```
{
  "RCSMessage" : {
    ...
    "msgId" : "...",
  },
  "reason" : "...",
  "file" : "...",
  "messageContact" : {...},
  "callbackURL" : "...",
  "event" : "..."
}
```

After the attack, the “RCSMessage” field of the request was removed.

3.2.6.1 Results of the experiment

The chaos logs show the operation of deleting “RCSMessage”:

```
RemoveFieldAttack Original request was {"RCSMessage" : {"msgId" : "...", ...}, "reason" : "...", "file" : "...", "messageContact" : {...}, "callbackURL" : "...", "event" : "..."} removed field is: "RCSMessage" : {"msgId" : "...", ...}
```

3.2.6.2 Evaluate the Hypothesis

After running this experiment, a *NullPointerException* can be found in the logs of the target system. By observing the source of the problem an unsafe operation was discovered (“payload” refers to the request body):

```
payload.getRcsMessage().getMsgId();
```

In the shown code snippet, “get...” methods can be seen being called on the object representing the request’s body. However, the first method call returns null in this case, due to the Remove Field Attack. Therefore, the next call will be executed on a null object, throwing a *NullPointerException*. The target system does not handle this error and responds with HTTP 500 and an empty response body, falsifying the hypothesis.

Proper field validation should be set up around this request and in case of missing required data an appropriate error response should be issued to the caller.

3.2.7 Scenario 7

In this experiment the target system expects a request having the following fields (the irrelevant fields for this experiment are omitted):

```
{
  ...
  "message":{
    ...
    "attributes":{
      ...
      "message_type" : "<type-from-predefined-list>"
    }
  }
}
```

3.2.7.1 Results of the experiment

The remove field attack caused the deletion of the “message_type” field from “attributes”. In the logs of the chaos tool the operation of removing the targeted field can be seen:

```
RemoveFieldAttack Original field was "attributes" :
{"message_type":"<type-from-predefined-list>", ...}", removed field
is: "message_type":"<type-from-predefined-list>"
```

3.2.7.2 Evaluate the Hypothesis

The logs of the target system showed a *NullPointerException* related to this experiment. Hence, the response to the caller was HTTP 500 and an empty body. In the target system, the cause of this exception was observed. The error came from a switch-case statement, where the evaluated object became null (due to removing the “message_type”).

Using non null-checked objects in switch-case statements is a bad programming practice, and proper validation should be set up to avoid this failure. Therefore, the hypothesis was falsified in this scenario.

4. Add Field Experiment

These experiments are designed to test the failure handling capabilities of the target system to HTTP requests containing extra fields in the POST body. These fields can be part of the request as a result of an unexpected change in the API.

4.1 Hypothesis

Target system is resilient to added extra fields in requests.

4.2 Experiment Design

For this experiment the add fields attack is activated and targeted to the controller. The attack configuration is set as follows (other fields set as false):

```
{
  "addFieldsActivated" : true,
  "attackFrequency" : 1,
  "isChaosPluginEnabled" : true
}
```

And the target configuration is set like this (other fields set as false):

```
{
  "controllerTargetActivated" : true
}
```

4.2.1 Scenario 1

For the first scenario the attack has been targeted to a request that expects the following valid schema:

```
{
  "to": "{{to}}",
  "message": {
    "type": "text",
    "text": " Hello from Add field 1!!!"
  }
}
```

4.2.1.1 Results of the experiment

After the attack execution, the chaos tool logs were inspected to verify the tool's capabilities to add fields:

```
AddFieldAttack Starting...
AddFieldAttack Intercepted Path: /rcs/v1/cZl8tXu5eQFNBZUF/messages
AddFieldAttack Original request:
{"to":"+36302*****","message":{"text":"Hello from Add field
1!!!","type":"text"}}
AddFieldAttack Added field: "chaos_extra_key":"chaos_extra_value"
AddFieldAttack Modified request:
{"chaos_extra_key":"chaos_extra_value","to":"+36302*****","message"
:{"text":"Hello from Add field 1!!!","type":"text"}}
AddFieldAttack Execution has ended
```

4.2.1.2 *Evaluate the Hypothesis*

Despite the attack execution, the request was successfully handled and no error messages were found in the target system's logs, therefore, the hypothesis was validated.

4.2.2 *Scenario 2*

For this scenario the attack has been targeted to a request that has the following schema:

```
{
  "subscription":"..",
  "message":{...}
}
```

4.2.2.1 *Results of the experiment*

After the attack was executed, the chaos tool logs were inspected and the results show the execution of the attack:

```
Chaos-plugin - AddFieldAttack Starting...
Chaos-plugin - AddFieldAttack Intercepted Path:
/rbm/cZl8tXu5eQFNBZUF/message_callback
Chaos-plugin - AddFieldAttack Original request:
{"subscription":"...", "message":{...}}
Chaos-plugin - AddFieldAttack Added field:
"chaos_extra_key":"chaos_extra_value"
Chaos-plugin - AddFieldAttack Modified request:
{"chaos_extra_key":"chaos_extra_value","subscription":"...", "message
":{...}}
Chaos-plugin - AddFieldAttack Execution has ended
```

4.2.2.2 *Evaluate the Hypothesis*

Similar to the first scenario, irrespective of the attack's ability to inject the extra fields, the request was successfully handled and no error messages were found, validating the hypothesis.

5. Fail Request Experiment

The experiments in this section are focused on testing the target system's resilience to failed requests. The goal here is to explore if the errors are handled when the target system expects an incoming request or a callback with a response body but never gets it.

5.1 Hypothesis

Target system is resilient to failed requests.

5.2 Experiment Design

In order to conduct experiments with the fail request attack, the chaos tool needs to be configured. The endpoints of the tool are used to enable and activate the attack:

```
{
  "failRequestActivated" : true,
  "attackFrequency" : 1,
  "isChaosPluginEnabled" : true
}
```

The target of the attack is set to REST endpoints (other fields set as false):

```
{
  "controllerTargetActivated" : true
}
```

5.2.1 Scenario 1

In the first scenario, an internal call is made in the target system, and the response is expected to have the following fields:

```
{
  "result" : ...,
  "imsi" : ...
}
```

The experiments here considered two different use cases of the target system making the same internal call. Even though the processes happened in different parts of the system, the results pointed to the same problem.

5.2.1.1 Results of the experiment

In the chaos logs the failed requests can be seen:

```
FailRequestAttack Intercepted Path: /lookup?msisdn=46070*****
```

```
FailRequestAttack Intercepted Path: /lookup?msisdn=43760*****
```

After running the experiments an interesting result can be found in the logs of the target system. There were attempts to handle the empty body and they threw exceptions in both cases:

```
ERROR - Permanent error for id <GUID> reason: Number lookup failed  
for msisdn 46070*****  
PermanentRejectException: Number lookup failed for msisdn 4607070798
```

```
ERROR - No Supplier for 43760***** with <GUID>  
ERROR - Permanent error for id <GUID> reason: No provisioned  
supplier for 43760*****  
PermanentRejectException: No provisioned supplier for 437600000
```

5.2.1.2 *Evaluate the Hypothesis*

In the previous log snippets, it can be observed that on the “ERROR” level, a properly configured censor mechanism is present to hide the phone numbers. However, the exception lacks this anonymization, and the target system logs this sensitive information.

Phone numbers are considered Personally Identifiable Information (PII) by the GDPR (General Data Protection Regulation) [32], hence they fall into the scope of this law. Article 5 of GDPR says “[PII should be] processed in a manner that ensures appropriate security of the personal data, including protection against unauthorized or unlawful processing and against accidental loss, destruction or damage, using appropriate technical or organizational measures” [32]. Hence, having phone numbers in long-term storage is not compliant with GDPR. Therefore, the hypothesis was falsified.

5.2.2 Scenario 2

In the second experiment requests that the target system expected as callback messages from MNOs were considered.

5.2.2.1 *Results of the experiment*

While conducting these experiments the fail request attack intercepted many different requests in the target system:

```
FailRequestAttack Intercepted Path: /<MNO1>/.../message_callback  
FailRequestAttack Intercepted Path: /<MNO2>  
FailRequestAttack Intercepted Path: /<MNO3>/.../messages
```

5.2.2.2 Evaluate the Hypothesis

As a summary of these experiments, it can be observed that the target system was resilient to the failure of these requests. It has an asynchronous manner of processing incoming messages hence waiting for these requests did not cause any time-out related issue. Therefore, the hypothesis was validated.

6. Creating High Memory Usage Experiment

The objective of this experiment is to determine whether the target system is resilient to high memory consumption scenarios. In a typical web server, there are many processes running in parallel that could have an impact on its resources, particularly memory. The experiment looks to evaluate the coping abilities of the target system in these types of settings.

6.1 Hypothesis

Target system is resilient to high memory usage.

6.2 Experiment Design

For this experiment the memory attack is activated. The chaos plugin configuration endpoints are used to make the attacks configuration as follows (other fields set as false):

```
{
  "memoryActivated" : true,
  "attackFrequency" : 1,
  "isChaosPluginEnabled" : true
  "attackFrequency" : "3"
}
```

And the target configuration as this:

```
{
  "controllerTargetActivated" : true,
  "serviceTargetActivated" : true,
  "repositoryTargetActivated" : true
}
```

6.2.1 Scenario 1:

For the first scenario, the attack has been targeted to a request made through the publicly available API (in this case the particular endpoint being hit is irrelevant).

6.2.1.1 Results of the experiment

After executing the attack, the chaos tool logs were inspected. These logs contain relevant information regarding memory usage. As such the max memory, used memory, and total free memory were obtained using Java's provided interface to query the application's environment [33]. These values can be described as follows:

- **Max Memory:** Returns the maximum amount of memory that the JVM will attempt to use [33].
- **Allocated Memory:** Returns the total amount of memory in the JVM [33]. This means the allocated space reserved for the Java process. The value returned by this method may vary over time.
- **Free Memory:** Returns the amount of free memory in the JVM [33].
- **Used Memory:** This is a calculated value and is the result of solving the following equation:

$$\text{Used Memory} = \text{Total Memory} - \text{Free Memory}$$

The following picture (Figure 5.) explains in a graphical way how the memory calculations are obtained:

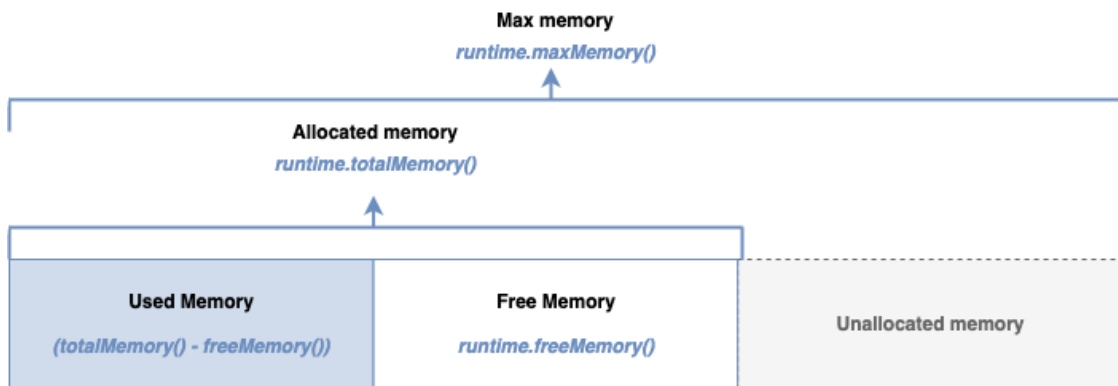


Figure 5. Memory Calculations in the JVM

Furthermore, the tool also calculates the total memory it has consumed over the accumulated time. This value is logged by the tool as follows:

- **Total eaten memory:** The total accumulated memory consumed by the tool up to the current cycle.

Having explained how those values are calculated, the logs obtained from the system can now be inspected. The following snippet shows the beginning, middle, and end values obtained after the attack was executed (other values were omitted for brevity):

```
ControllerAspect Intercepted Class: '...controller.AController',
Method: 'index'
MemoryAttack Starting...

HighMemoryThread Max Memory: 3,817,865,216.00
HighMemoryThread Allocated Memory: 1,058,537,472.00
HighMemoryThread Used memory: 778,153,632.00
HighMemoryThread Total eaten memory: 0.00
(...)
HighMemoryThread Max Memory: 3,817,865,216.00
HighMemoryThread Allocated Memory: 2,585,264,128.00
HighMemoryThread Used memory: 1,756,268,168.00
HighMemoryThread Total eaten memory: 1,876,086,732.00
(...)
HighMemoryThread Max Memory: 3,817,865,216.00
HighMemoryThread Allocated Memory: 3,850,895,360.00
HighMemoryThread Used memory: 2,564,202,296.00
HighMemoryThread Total eaten memory: 2,681,393,100.00

HighMemoryThread Memory space exhausted
HighMemoryThread Releasing memory space
HighMemoryThread Execution has ended
```

6.2.1.2 Evaluate the Hypothesis

After executing the experiment, it can be confirmed that the attack was effective at consuming the memory resources of the application. At the beginning of the attack, the total free memory was: 3,039,711,584. The total memory consumed after the experiment was 2,681,393,100. The allocated memory also showed to be growing as the attack progressed moving from 1,058,537,472.00 to 3,850,895,360.00. Moreover, these final values bordered the limit of the max memory which was: 3,817,865,216. At this point, the attack ends, as this will ensure that the JVM can release the newly allocated memory.

Several experiments were executed with different values for the memory factor property, however, similar results were obtained. Hence despite the high memory usage reported by the tool, the request was properly handled, and a successful response was received, validating the hypothesis.

7. Creating high CPU usage Experiment

This experiment looks to validate the resilience of the target system to high CPU usage. The target system should be able to tolerate multiple processes running at different CPU levels and show its resilience by gracefully coexisting with them. This experiment looks to evaluate the ability of the target system to succeed in such circumstances.

7.1 Hypothesis

Target system is resilient to high CPU usage.

7.2 Experiment Design

For this experiment the CPU attack is activated using the configuration endpoints, the final values look like this: (other properties set as false):

```
{
  "cpuActivated" : true,
  "cpuAttackFactor" : 300,
  "isChaosPluginEnabled" : true,
  "attackFrequency" : "2"
}
```

And the target configuration is set as this:

```
{
  "controllerTargetActivated" : true,
  "serviceTargetActivated" : true,
  "repositoryTargetActivated" : true
}
```

Prior to running the attack, the CPU usage was monitored against its stable state for comparison purposes. The target system was inspected while processing a normal request with no attacks. Using the Unix system's "top" command [34], the CPU usage on the system can be monitored.

The %CPU or CPU usage shows the percentage of CPU that is being used by the process [34]. On multi-core systems, this percentage can be greater than 100% [34]. In this case, the staging system dual-core processor (as described in the environment section 5.4.1), this means that the maximum CPU that can be reported by the tool for this environment is 200%.

The top command provides a dynamic real-time view of the processes running in the system [34]. The top portion contains the header information. The lower part contains the list of the running processes and the statistics of resource usage.

```
PID    %CPU   COMMAND
29318  2.0    java
29318  20.2   java
29318  20.4   java
29318  5.6    java
```

The system shows a CPU percentage that oscillates between 2.0% and 20% but not beyond this threshold.

7.2.1 Scenario 1

For this scenario, the request was executed against one of the public endpoints (similar to the memory attack, in this scenario the particular endpoint being hit is not relevant).

7.2.1.1 Results of the experiment

After running the attack, the chaos tool logs were inspected. These show that the tool was able to effectively target the @Service, @Repository, and @Controller annotated classes. For brevity a smaller sample of the logs is presented:

```
CPUAttack Starting...

ServiceAspect Intercepted Class: '...service.AService', Method:
'generate'

RepositoryAspect Intercepted Class: '...repository.ARepository',
Method: 'getMessageId'

ControllerAspect Intercepted Class: '...controller.AController',
Method: 'messageCallback'

HighCPUThread A new thread id: 15934 is running
HighCPUThread Execution for thread id: 15934 has ended
```

A total of fourteen separate threads were created as a result of running this experiment. Using the same Unix system's "top" command previously presented, the monitored values obtained from the target system are:

```
PID    %CPU  COMMAND
29318  5.6   java
29318  175.4 java
29318  197.7 java
29318  200.0 java
29318  166.1 java
29318  81.1  java
29318  24.1  java
29318  2.0   java
```

7.2.1.2 Evaluate the Hypothesis

The monitored values show a spike in the CPU usage, the percentage climbs all the way to 200% at the peak of the attack. This means that the maximum CPU usage was reached as a result of running both of the CPU cores at 100% capacity. As the threads finish their execution and resources are freed the monitored values show how the usage drops to normal levels of close to 2% and oscillating.

Additionally, an interesting finding was discovered regarding response times. As the CPU percentage escalated, so did the response times, with responses taking almost one minute and a half, yet being able to gracefully handle the request. This metric was not considered in the initial experiment design, nonetheless, it is a finding that shows the chaos capabilities of the tool.

Other scenarios with higher CPU factors were explored with similar results, hence this scenario was chosen as a sample representation of the group.

Regardless of the high CPU usage caused by the chaos tool and the otherwise long response times the request was properly handled, and a successful response was received, therefore the hypothesis was validated.

8. Stop all running attacks Experiment

In these experiments, the intent is to demonstrate the tool's ability to stop all running attacks when the chaos tool is disabled. This means that no new perturbations are introduced as a result of deactivating the tool and that attacks running in self-iterating loops are short-circuited.

These experiments are different from the previously presented experiments as they look to demonstrate a feature of the tool as opposed to the target system's resilience. The hypothesis looks to validate whether the tool behaves as defined in requirement 5 in section 5.3.2.

8.1 Hypothesis

The tool is able to stop running attacks.

8.2 Experiment Design

For this experiment the tool was deactivated using the provided endpoint:

```
POST /chaos-plugin/disable
```

8.2.1 Scenario 1

For this scenario the modify, create and add field attacks are activated using the configuration endpoints, the configuration looks like this (other properties set as false):

```
{
  "modifyFieldsActivated": true,
  "addFieldActivated": true,
  "removeFieldActivated": true,
  "isChaosPluginEnabled" : true
}
```

And the target configuration is set as this:

```
{
  "controllerTargetActivated" : true,
  "serviceTargetActivated" : false,
  "repositoryTargetActivated" : false
}
```

A request is executed against the target system (the particular request being sent is irrelevant)

8.2.1.1 Results of the experiment

After executing the request, the chaos tool logs were inspected. The results confirm that no additional logs were made. Moreover, a successful response was received. This means that no attacks were executed.

8.2.1.2 Evaluate the Hypothesis

The results show that the tool did not execute any of the attacks due to its disabled status. This demonstrates the tool's ability to stop the introduction of new attacks and validates the hypothesis.

8.2.2 Scenario 2

For this scenario the memory attack is activated. This attack falls in the self-iterating category. This scenario looks to validate the short-circuiting capabilities of the tool. Therefore the configuration is set like this (other properties set as false):

```
{
  "memoryActivated": true
}
```

And the target configuration is set as this:

```
{
  "controllerTargetActivated" : true,
  "serviceTargetActivated" : false,
  "repositoryTargetActivated" : false
}
```

A request is made against the target system (the particular request is irrelevant in this scenario) and at some point in the execution, the disable endpoint is called.

8.2.2.1 Results of the experiment

After executing the request and hitting the disable endpoint, the chaos tool logs were inspected. The results show how the execution was halted and the memory attack was ended:

```
MemoryAttack Starting...
HighMemoryThread Max Memory: 3,817,865,216.00 Allocated Memory:
1,014,497,280.00
HighMemoryThread Used memory: 319,810,416.00 Total Free memory:
3,498,054,800.00
HighMemoryThread Limit: 3,626,971,955.00
HighMemoryThread Total eaten memory: 268,435,456.00
HighMemoryThread Chaos disabled: Execution halted
```

8.2.2.2 *Evaluate the Hypothesis*

The results show that the tool was able to halt the execution of the attack before it had ended. These findings prove the tool's ability to stop all running attacks, validating the hypothesis.

References

- [1] Heorhiadi V., Rajagopalan S., Jamjoom H., Reite M., Sekar V. “Gremlin: Systematic Resilience Testing of Microservices.” 2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS).
- [2] Monperrus, M. “Principles of Antifragile Software.” Proceeding Programming '17 Companion to the first International Conference on the Art, Science and Engineering of Programming Article No. 32.
- [3] Alvaro P., Andrus K., Sanden C., Rosenthal C, Basiri A., Hochstein L., “Automating Failure Testing Research at Internet Scale”, Proceedings of the Seventh ACM Symposium on Cloud Computing, October 05-07, 2016, Santa Clara, CA, USA
- [4] Goldman D. (2011) “Why Amazon's cloud Titanic went down”. Available online: https://money.cnn.com/2011/04/22/technology/amazon_ec2_cloud_outage/index.html (*Last visited: 2019.12.11*)
- [5] McNamara P. (2009) “Missing dot drops Sweden off the Internet.” Available online: <https://www.networkworld.com/article/2232047/missing-dot-drops-sweden-off-the-internet.html> (*Last visited: 2019.12.11*)
- [6] Zhang L., Morin B., Haller P., Baudry B., Monperrus M. “A Chaos Engineering System for Live Analysis and Falsification of Exception-handling in the JVM” IEEE Transactions on Software Engineering.
- [7] Tucker H. O., Hochstein L., Jones N., Basiri A., Rosenthal C. “The Business Case for Chaos Engineering.” *IEEE Cloud Computing* 5 (2018): 45-54.
- [8] J. Robbins, “Resilience Engineering: Learning to Embrace Failure,” ACM Queue, vol. 10, no. 9, 2012.
- [9] Izrailevsky Y., Tseitlin A. (2011) “The Netflix simian army”. Available online: <http://techblog.netflix.com/2011/07/netflix-simian-army.html>. (*Last visited: 2019.12.11*)
- [10] Basiri A., Behnam N., de Rooij R., Hochstein L., Kosewski L., Reynolds J., Rosenthal C. “Chaos engineering” *IEEE Software*, 33(3):35–41, May 2016.
- [11] Nakama H. “Inside Azure search: Chaos engineering.” Available online: <https://azure.microsoft.com/en-us/blog/inside-azure-search-chaos-engineering/>, July 2015. (*Last visited: 2019.12.11*)
- [12] Hof R. Interview: “How Facebook's Project Storm Heads Off Data Center Disasters” 2016. Available online. <https://www.forbes.com/sites/roberthof/2016/09/11/interview-how-facebooks-project-storm-heads-off-data-center-disasters/> (*Last visited: 2019.12.11*)
- [13] Blohowiak A., Basiri A., Hochstein L., Rosenthal C. “A Platform for Automating Chaos Experiments.” *2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)* (2016): 5-8.
- [14] Simonsson J., Zhang L., Morin B., Baudry T., Monperrus M. “Observability and Chaos Engineering on System Calls for Containerized Applications in Docker” 2019. KTH Royal Institute of Technology, Sweden.
- [15] Peffers K., Tuuranen T., Rothenberg M., Chatterjee S. “A Design Science Research Methodology for Information Systems Research” 2007. *Journal of Management Information Systems*. 24:3, 45-77, DOI: 10.2753/MIS0742-1222240302
- [16] Will S. “Improve application resiliency with chaotic testing” Available online: https://www.ibm.com/garage/method/practices/manage/practice_chaotic_testing/ (*Last visited: 2019.12.11*)

- [17] Rosenthal C., Blohowiak A., Hochstein L., Jones N., Basiri A. „Chaos engineering - Building confidence in system behavior through experiments” O’Reilly, 2017.
- [18] Chaos Community (2018) “Principles of chaos engineering”. Available online. <https://principlesofchaos.org/?lang=ENcontent> (*Last visited: 2020.03.25*)
- [19] Butow T. (2019) “Chaos Engineering: the history, principles, and practice” Available online: <https://www.gremlin.com/community/tutorials/chaos-engineering-the-history-principles-and-practice/> (*Last visited: 2019.12.11*)
- [20] M. Conti, N. Dragoni and V. Lesyk, "A survey of man in the middle attacks", IEEE Commun. Surveys Tuts., vol. 18, no. 3, pp. 2027-2051, 3rd Quart. 2016.
- [21] Hombergs T., “Quick Guide to Building a Spring Boot Starter”, Available online: <https://reflectoring.io/spring-boot-starter/> (*Last visited: 2020.03.12*)
- [22] VMware, Inc. or its affiliates (2020) “Creating your own Auto-Configuration”, Available online: <https://docs.spring.io/spring-boot/docs/2.1.12.RELEASE/reference/html/boot-features-developing-auto-configuration.html> (*Last visited: 2020.05.18*)
- [23] VMware, Inc. or its affiliates (2020) “ClientHttpRequestInterceptor”, Available online: <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/http/client/ClientHttpRequestInterceptor.html> (*Last visited: 2020.05.18*)
- [24] Oracle and/or its affiliates (2019). “Pausing Execution with Sleep”, Java Documentation. Available online. <https://docs.oracle.com/javase/tutorial/essential/concurrency/sleep.html> (*Last visited: 2020.04.22*)
- [25] Oracle and/or its affiliates (2018). “Class Thread”, Java Documentation. Available online. <https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html> (*Last visited: 2020.04.21*)
- [26] Mozilla and individual contributors (2020). “Working with JSON”, MDN web docs. Available online: <https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/JSON> (*Last visited: 2020.04.22*)
- [27] Oracle (2020). “The Essentials of Filters”. Available online <https://www.oracle.com/java/technologies/filters.html> (*Last visited: 2020.05.22*)
- [28] Oracle and/or its affiliates (2019). “What are Regular Expressions?” Available online <https://docs.oracle.com/javase/tutorial/essential/regex/intro.html> (*Last visited: 2020.05.22*)
- [29] VMware, Inc. or its affiliates (2020) “Aspect Oriented Programming with Spring”, Available online: <https://docs.spring.io/spring/docs/2.5.x/reference/aop.html> (*Last visited: 2020.03.12*)
- [30] Ding Yuan, Yu Luo, Xin Zhuang, Rodrigues G. R., Xu Zhao, Yongle Zhang, Jain P., Stumm M. “Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems” In 11th USENIX Symposium on Operating Systems Design and Implementation, pp. 249–265, 2014
- [31] Hornsby A. (2019) “Chaos Engineering”, Available online: <https://medium.com/@adhorn/chaos-engineering-ab0cc9fbd12a> (*Last visited: 2020.05.05*)
- [32] “General Data Protection Regulation” (2018), Available Online: <https://gdpr-info.eu> (*Last visited: 2020.05.05*)
- [33] Kerrisk M (2020). “Top, User Commands” Linux man pages. Available online: <http://man7.org/linux/man-pages/man1/top.1.html> (*Last visited: 2020.05.06*)
- [34] Oracle and/or its affiliates (2020). “Class Runtime”. Available online: <https://docs.oracle.com/javase/8/docs/api/java/lang/Runtime.html> (*Last visited: 2020.05.06*)